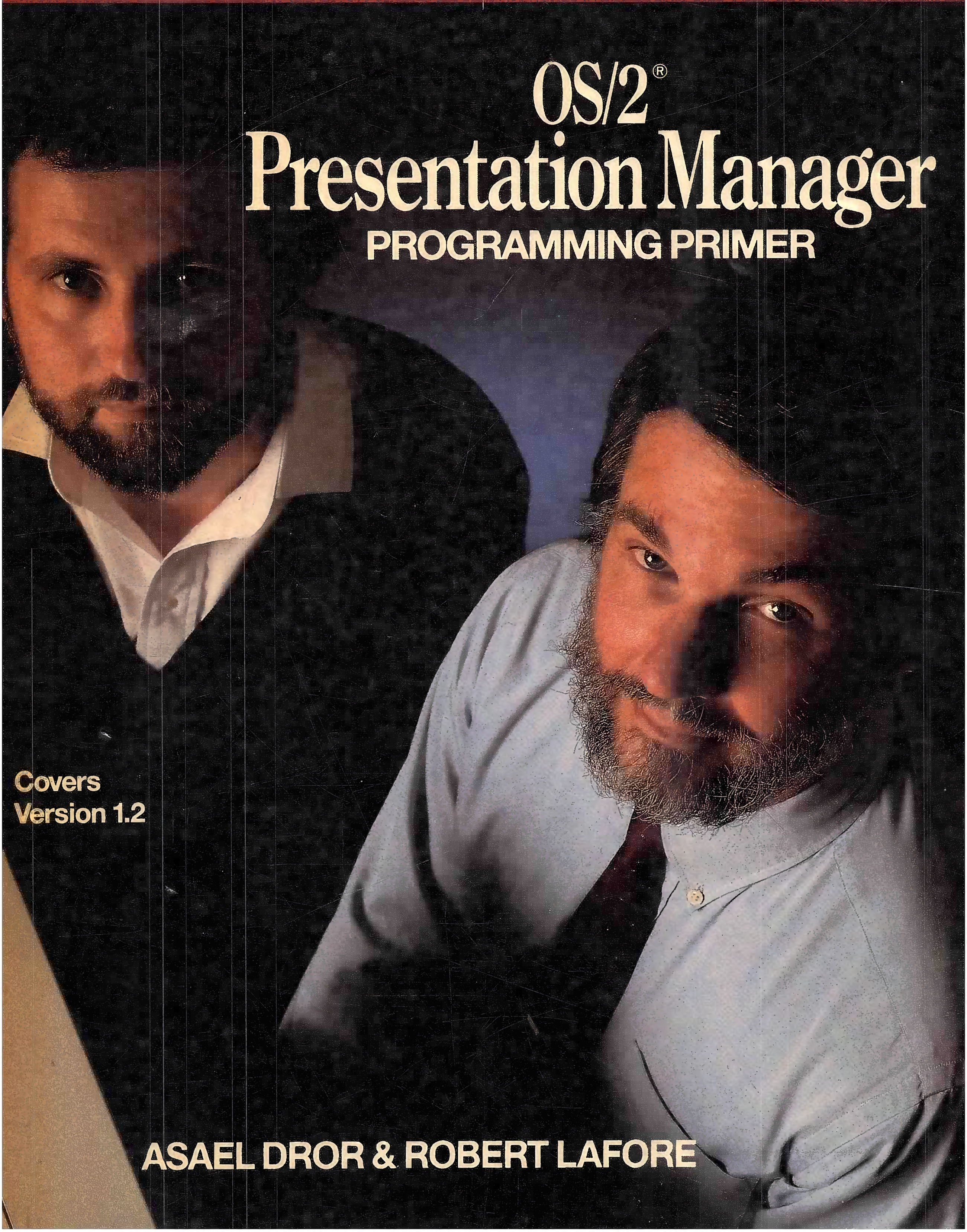
Osborne McGraw-Hill

OS/2®
Presentation Manager
PROGRAMMING PRIMER



Covers
Version 1.2

ASAEL DROR & ROBERT LAFORE

OS/2® PRESENTATION MANAGER PROGRAMMING PRIMER

OS/2® PRESENTATION MANAGER PROGRAMMING PRIMER

Asael Dror and Robert Lafore

Osborne **McGraw-Hill**

Berkeley New York St. Louis San Francisco
Auckland Bogotá Hamburg London Madrid
Mexico City Milan Montreal New Delhi Panama City
Paris São Paulo Singapore Sydney
Tokyo Toronto

Osborne **McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations and book distributors outside of the U.S.A., please write to Osborne **McGraw-Hill** at the above address.

A complete list of trademarks appears on page 605.

OS/2® Presentation Manager Programming Primer

234567890 DOC 90

ISBN 0-07-881467-7

This book is dedicated with love to Rachel and Yehezkel

—Asael Dror

This book is dedicated to Beca and Tenaya, the point of the exercise

—Robert Lafore

CONTENTS AT A GLANCE

CONTENTS

ACKNOWLEDGMENTS

An amazingly large number of people have contributed to the creation of this book. We are indebted to them all.

Scott Ludwig and Byron Dazey, from Microsoft, provided invaluable and highly detailed technical feedback.

Mark Mackaman, Scott Brooks, Cameron Myhrvold, and Alan Cobb provided documentation and beta versions of the software, without which this book could not have been written.

Osborne/McGraw-Hill authors William Murray and Chris Pappas offered helpful comments and insights on the manuscript.

Jeff Pepper, senior editor at Osborne/McGraw-Hill, has been a patient and very able overseer of the entire project; we admire his unfailing good humor, ability to juggle a hundred chapters at once, and expertise as a sysop. His assistant Judith Brown was a charming and able recipient of our telephoned corrections.

Our special thanks to our copy editor, Jan Jue. She is one of the world's premier editors, with a light touch on our deathless prose and an uncanny ability to pinpoint inconsistencies in the content.

In the Osborne/McGraw-Hill production department, Dusty Bernard has shouldered the millions of details involved. Our hats are also off to the entire typesetting department.

The authors' names are listed in alphabetical order; they contributed equally to the book.

PREFACE

It is becoming clear that OS/2 will be the dominant operating system throughout the 1990s. Its large address space, multitasking and networking capabilities, and the Presentation Manager—OS/2's graphical user interface—have put it far ahead of other microcomputer operating systems. Its adherence to IBM's Systems Applications Architecture establishes it as a new standard in the corporate marketplace. Programmers will need to examine its capabilities closely, and most—either now or in the near future—will need to write applications for it.

Presentation Manager programming has acquired the reputation of being a complex and arcane subject with a long learning curve. Our intention in this book is to prove that PM isn't so hard after all. By focusing on the fundamentals and explaining in detail just how each PM feature works, we shorten the learning curve and show that PM is no harder than, say, learning a new computer language.

We start slowly, using short, easy-to-follow C language programming examples, with figures and output screens to illustrate the concepts. Gradually we work our way up to more advanced topics. When you finish this book you should be ready to write almost any Presentation Manager application.

There are two parts to OS/2: the kernel and the Presentation Manager. The kernel performs functions that are mostly hidden from the user, such as disk I/O, memory allocation, and multitasking. The work of the Presentation Manager, on the other hand, is there on the screen for all to see. It manages windows, menus, dialog boxes, radio buttons, and the other ele-

ments of the graphics interface. The Presentation Manager is the glamorous part of OS/2.

This book teaches you how to write programs for the Presentation Manager. You'll learn how to create and manage all the aspects of the user interface. You'll also learn a new style of programming: "message based," as opposed to traditional "procedure-based" programs.

The Presentation Manager also contains a rich collection of graphics functions. We explain how to use them, so you can draw lines, circles, and other graphics elements, and display text in different fonts. In addition, we show you how to exchange data with other applications, how to use multitasking in Presentation Manager programs, how to use the concepts of object-oriented programming to simplify your programs, and much more.

For most programmers, learning to program the Presentation Manager will be a radically new experience. We hope that in this book we manage to convey some of the sense of fascination and excitement we felt in our first encounter with the Presentation Manager.

You can order a disk containing source and executable code for the example programs in this book. This can save considerable typing. Use the order form at the end of this preface, or contact

Wisdom Software Inc.
P.O. Box 460310
San Francisco CA 94146-0310

A.D. and R.L.

ADDITIONAL HELP FROM OSBORNE/MCGRAW-HILL

Osborne/McGraw-Hill provides top-quality books for computer users at every level of computing experience. To help you build your skills, we suggest that you look for the books in the following Osborne/M-H series that best address your needs.

The "Teach Yourself" Series is perfect for beginners who have never used a computer before or who want to gain confidence in using program basics. These books provide a simple, slow-paced introduction to the fundamental usage of popular software packages and programming languages. The "Mastery Learning" format ensures that concepts are learned thoroughly before progressing to new material. Plenty of exercises and examples (with answers at the back of the book) are used throughout the text.

The "Made Easy" Series is also for beginners or users who may need a refresher on the new features of an upgraded product. These in-depth introductions guide users step-by-step from the program basics to intermediate-level usage. Plenty of "hands-on" exercises and examples are used in every chapter.

The "Using" Series presents fast-paced guides that quickly cover beginning concepts and move on to intermediate-level techniques, and even some advanced topics. These books are written for users who are already familiar with computers and software, and who want to get up to speed fast with a certain product.

The "Advanced" Series assumes that the reader is already an experienced user who has reached at least an intermediate skill level and is ready to learn more sophisticated techniques and refinements.

"The Complete Reference" is a series of handy desktop references for popular software and programming languages that list every command, feature, and function of the product along with brief, detailed descriptions of how they are used. Books are fully indexed and often include tear-out command cards. "The Complete Reference" series is ideal for all users, beginners and pros.

"The Pocket Reference" is a pocket-sized, shorter version of "The Complete Reference" series and provides only the essential commands, features, and functions of software and programming languages for users who need a quick reminder of the most important commands. This series is also written for all users and every level of computing ability.

The "Secrets, Solutions, Shortcuts" Series is written for beginning users who are already somewhat familiar with the software and for experienced users at intermediate and advanced levels. This series gives clever tips and points out shortcuts for using the software to greater advantage. Traps to avoid are also mentioned.

Osborne/McGraw-Hill also publishes many fine books that are not included in the series described above. If you have questions about which Osborne book is right for you, ask the sales person at your local book or computer store, or call us toll-free at 1-800-262-4729.

The Editor

OTHER OSBORNE/MCGRAW-HILL BOOKS OF INTEREST TO YOU

We hope that OS/2 Presentation Manager Programming Primer will assist you in mastering this fine product and will also encourage you to learn more about other ways to better use your computer.

If you're interested in expanding your skills so you can be even more computer efficient, be sure to take advantage of Osborne/M-H's large selection of top-quality computer books. Here are just a few books that complement *OS/2 Presentation Manager Programming Primer.*

OS/2 Programming: An Introduction by Herbert Schildt is a fast-paced guide that gets you up to speed on OS/2 version 1.1 intermediate-level programming techniques. A background in assembly language programming is not a prerequisite, although many example C programs are used in the text. Applications are emphasized.

OS/2 Programmer's Guide, Second Edition, Volume 1, written by Ed Iacobucci, leader of the IBM OS/2 design team, offers an in-depth introduction to OS/2 through version 1.1. It presents a complete overview of the OS/2 operating system and Presentation Manager.

OS/2 Programmer's Guide, Second Edition, Volume 2, also by Ed Iacobucci, is written for experienced OS/2 programmers and provides comprehensive coverage of the OS/2 API structure of version 1.1 and advanced multitasking. Appendixes cover OS/2 function calls, the family API, OS/2 error codes, linker control statements, and sample programs.

Assembly Language Programming Under OS/2, by Bill Murray and Chris Pappas, is a hands-on guide that serves as an excellent introduction to OS/2 version 1.1 assembly language programming.

The Editor

DISK ORDER FORM

Please send me ______ copies, at $25.00 each, of the companion disk for *OS/2 Presentation Manager Programming Primer.* California residents add $1.63 sales tax. Outside the U.S. and Canada, add $5.00 for shipping and handling. The disk contains source and executable code for all the example programs in the book, as well as the binary files for all icons, pointers, and bitmaps.

Please print:

Name ____

Title ______________________ **Phone** ______________________

Organization ____

Address ____

City ______________________ **State** __________ **ZIP** ____________

Disk size: 3-1/2" _______ **or 5-1/4"** _______

Payment method:

Check enclosed _______ **(payable to Wisdom Software)**

VISA _______ **MasterCard** _______ **American Express** _______

Card no. ______________________________ **Exp. date** ____________

Signature ____

Send to: Wisdom Software Inc.
P.O. Box 460310
San Francisco CA 94146-0310
U.S.A.

Osborne/McGraw-Hill assumes NO responsibility for this offer. This is solely an offer of Wisdom Software Inc., and not of Osborne/McGraw-Hill. Please allow four to six weeks for delivery.

WHY THIS BOOK IS FOR YOU

Once every decade or so the accepted programming environment undergoes a fundamental change. In the last such change, CP/M was replaced by MS-DOS. Now, another revolution is coming: MS-DOS will be supplanted by OS/2 and the Presentation Manager—its graphical user interface. If you know how to program the PM, you will be ready to cash in on this revolution.

This book shows you how to program PM's user interface, and also teaches PM graphics. It is simply and clearly written, and assumes no prior experience with OS/2. All you need is some knowledge of the C programming language. Numerous C language examples demonstrate each feature described.

Covering the latest 1.2 version of PM, this book is up-to-date and authoritative. It will have you writing full-scale Presentation Manager applications in a few short months.

LEARN MORE ABOUT OS/2 PRESENTATION MANAGER

Here is another excellent Osborne/McGraw-Hill book on OS/2 Presentation Manager that will help you build your skills and maximize the power of the operating system you have selected.

If you're already programming OS/2, see *OS/2 Presentation Manager Graphics: An Introduction* (Pappas and Murray) for a guide to unlocking the power of Presentation Manager graphics. The book starts with basic concepts and a discussion of individual Presentation Manager commands and then builds to increasingly complex programs for line, bar, and pie charts.

The Editor

PART I: INTRODUCTION

1

ABOUT THIS BOOK

In this chapter we'll explain what the book is about and how it's organized. We'll also tell you what you need to know to use it and the hardware and software you'll need to compile and run the example programs.

WHAT YOU'LL LEARN FROM THIS BOOK

This book explains how to program the Presentation Manager (PM). PM is the graphics user interface that comes with the OS/2 operating system. It allows the user to interact with application programs by manipulating windows, pull-down menus, dialog boxes, and other visual elements. This interface replaces the old teletype-based interface of MS-DOS programs, in which the application and the user communicated by typing at each other. The PM also supplies a powerful set of graphics functions and other services to the programmer.

The API Functions

The programmer controls the elements of PM by calling functions built into a dynamic link library (DLL) provided with OS/2. These functions constitute what is called the *Applications Programming Interface* (*API*). There are

almost 500 of these functions available for PM programmers. Those that deal with the user interface begin with the letters "Win" (for Window), as in WinBeginPaint and WinCreateWindow. Those concerned with graphics (lines, circles, and so on) begin with the letters "Gpi" (for Graphics Programming Interface), as in GpiBox and GpiFullArc. We'll also be concerned with a few functions starting with "Dev" (for Device).

To program PM you need to learn how these functions work, and one of the purposes of this book is to familiarize you with the API functions. (You don't need to learn all of them. We cover the major API functions, but some are for specialized situations that will be mentioned only briefly, and others are close cousins of those we explain.)

However, learning about the API functions is only part of the story.

Message-based Program Architecture

For reasons to be explored in the following chapters, the structure of PM programs is different from that of traditional MS-DOS programs. Different components of the program communicate with the operating system, and with each other, by sending messages. We'll explain what this means and show many examples.

Resources and Other PM Building Blocks

Another element not used in traditional MS-DOS programs is the *resource.* A resource starts as a file, separate from your program, that contains data your program will use. It's then added to your program's .EXE file (or placed in a DLL file where your program can use it). A typical use for a resource is to hold the text of the menu selections ("File", "Open", "Close", and so on) used by your program.

Program Syntax and Creation

A typical PM program doesn't look like a standard C program. For instance, data may be declared to be of type HAB or type QMSG, rather than *unsigned int* or another normal C data type. Also, variable names themselves often look odd. A programmer might use the variable names *pchMsg* or *szString,* for example, instead of simply *Msg* or *String.* These and other aspects of program syntax make PM programs appear strange at first, but we'll demystify this syntax and explain the reasons for it.

We'll show how resources and definition files are used, and explain how the different elements of a program are combined into a working whole.

Utility Programs

Various utility programs are helpful for automating different tasks in PM programming. For instance, when we describe dialog boxes, we'll introduce you to DLGBOX, the dialog box editor. This utility automatically creates a resource for a dialog box and saves you from typing in the numerical coordinates of each radio button and caption in the box.

We'll describe other utilities as we go along, such as the icon editor.

HOW THIS BOOK IS ORGANIZED

This book is an introduction to programming the Presentation Manager. It is intended to be read from front to back; it is not a reference.

The book is divided into four parts. Part I gives an overall view of the Presentation Manager and shows how it relates to the OS/2 kernel and to the rest of the world. Part II is the "meat and potatoes" of the book. It begins by describing program syntax and construction. It goes on to explore all the aspects of the PM user interface, such as windows, menus, and dialog boxes, and the new program structures used to accommodate them.

Part III covers the second major aspect of the Presentation Manager: the functions used to draw graphics elements such as circles, lines, and boxes. It also shows how these elements can be combined to form complete pictures that can be stored, written to disk, and "played" to re-create the graphics image. We also cover the use of text on the graphics screen, including the use of different fonts.

Part IV of the book covers more global PM programming topics: how to exchange information between PM applications and how to use multitasking with PM programs.

WHAT YOU NEED TO KNOW BEFORE YOU START

Because PM programming is so different from traditional programming, you need surprisingly little background to begin learning it. Here's a brief list.

The C Language

There is only one firm prerequisite for reading this book: you should have some experience in the C programming language. C is the most popular language currently used for PM programming. All the example programs in this book are written in C, and the documentation from Microsoft and IBM, the creators of OS/2, is primarily in C.

Graphics User Interfaces

If you have never used a system with a graphics-based user interface, like Microsoft Windows or the Macintosh, you'll find it's easy to learn by experimenting with the PM itself and with PM applications. In a short time you'll see what PM does and what applications will need to do to fit into the PM environment.

What You Don't Need to Know

Surprisingly, you don't need to know how to program the OS/2 kernel to program the PM. We use a handful of kernel functions, but they're not central to PM programming, and we'll explain them as they arise. Of course, it doesn't hurt to know something about the kernel. Eventually, to write full-scale applications that make use of all of OS/2's features, you will need to understand both the kernel and the PM.

Experience in programming Microsoft Windows is definitely helpful, since the approach is basically the same for Windows and PM. However, there are many differences as well. This book does not presuppose any Windows programming experience.

Previous programming experience in MS-DOS is also not necessary, and not even very relevant. PM is a whole new world.

NECESSARY HARDWARE

This section discusses the hardware necessary to run OS/2 with the Presentation Manager, and to compile and run the programs in this book.

Computer

You'll need a computer capable of running OS/2. This means a machine based on an Intel 80286, 80386, or 80486 microprocessor. The 80286 is perfectly adequate.

Not all computers are compatible with OS/2. Because OS/2 bypasses the BIOS routines, it is far more hardware dependent than MS-DOS. If you're shopping for a new computer, make sure it runs OS/2 before you buy it. As time goes by, more and more computer vendors are supplying their own customized versions of OS/2, guaranteed to run on their machines.

Memory

We recommend 4 megabytes (MB) of memory. Actually, you can make do with 3 MB, but you won't have much flexibility multitasking. More than 4 MB is necessary only if you plan to run many large applications at the same time.

Hard Disk

You'll need a hard disk; OS/2 is too big to run on a floppy system, and it would be too slow, anyway. Once you've stored OS/2, the C compiler, and other odds and ends on your hard disk, you'll have used up 10 to 15 MB. Thus you should have a 30 MB hard disk at a minimum, and the bigger the better. Even more important than size is access time. OS/2 is disk-intensive, so the faster your hard disk, the happier you'll be.

You'll need a high-density diskette drive to read in the system disks: either the 5 1/4-inch 1.2 MB drive, or the PS/2-style 3 1/2-inch 1.44 MB drive.

Graphics

You can use a variety of graphics adapters for OS/2, including EGA and VGA. Many adapters work with OS/2, but check before you buy; some are incompatible. Since PM is graphics based, you obviously can't use a text-only display system. Anyway, such systems are history.

Mouse

You'll also need a mouse. While Microsoft and IBM recommend writing PM applications so they can be used with either keyboard or mouse input, the mouse is the preferred device. If you're going to develop PM programs, you'll need to learn how to program the mouse. Many of the examples in

this book are mouse-based. If you don't have much experience with a mouse, try it before you criticize it. You may be surprised at how quickly it becomes an indispensable input medium.

NECESSARY SOFTWARE

Here's a brief summary of the software you'll need to compile and run the programs described in this book.

The OS/2 Operating System

You'll need OS/2 itself. This book was written for version 1.2 but will work with version 1.1. (The earlier version, 1.0, is no longer supported; it contained only the kernel API calls and did not include the Presentation Manager.) You can use IBM's Extended Edition of OS/2, which contains database and communications support, but these extra features are not necessary for this book.

C Compiler

For software development you'll need a C compiler. This must be a compiler designed to generate protected-mode OS/2 programs; older compilers designed only for MS-DOS won't work. The Microsoft Optimizing compiler, version 5.1 or later, is the standard for OS/2 development. This is the compiler used for the examples in this book and described in the text. However, other vendors also produce OS/2-ready compilers that should work in substantially the same way. You'll also need a linker, which is usually included with the compiler.

The MAKE utility, also usually included with the compiler, is useful. Make files appear in the programming examples, but you can type in the relevant command lines by hand if you don't have MAKE.

Editor

Your source code editor should run in the protected mode of OS/2. It is possible to write programs using an MS-DOS editor running in the compatibility box, but this is a clumsy approach. Microsoft makes a protected-mode

editor, and more vendors are coming out with protected-mode versions of MS-DOS editors. An editor that takes advantage of the PM visual interface is convenient.

Development Kits

Microsoft and IBM market various products for OS/2 software development. The contents of these kits change frequently, so rather than try to describe them, we'll list those components that you need to use this book. These components may come with the compiler, in a developer's kit, or they may be available somewhere else.

Files

You'll need the file OS2.LIB. This doesn't contain the actual API routines, but contains information your program uses to make external references to these functions, for both the kernel and the PM. The routines themselves are in DLL files or in the OS/2 kernel. You'll also need a set of about a dozen header files, such as OS2.H, OS2DEFS.H, BSEDOS.H, and PMWIN.H. These contain function prototypes, structure definitions, data typedefs, and other elements necessary for program development.

Resource Compiler

PM programs make extensive use of resources, data used by your program and incorporated into your program's .EXE file (or into a DLL) using a special format. To compile this data you need a *resource compiler,* called RC in the Microsoft development system.

Other Utility Programs

Several utility programs are needed for specific tasks. There is ICONEDIT, for creating icons, pointers, and bitmaps; and DLGBOX, for creating dialog boxes. You may also want to use FONTEDIT to create or change text fonts.

Debuggers

A debugger is a key tool in program development. Most of us have used the ancient DEBUG or one of the newer debuggers like Microsoft's CodeView. CodeView has recently appeared in a version that handles multiple threads, so it is a good choice for OS/2 kernel programming. Using CodeView with two displays can be a powerful approach in PM programs.

Unfortunately, as of this writing, there is no debugger specifically designed to handle the PM architecture. Because the PM is message-based, it needs a debugger that will allow the trapping of messages and other activities particular to this environment. Until such a debugger appears, you'll have to make do with existing debuggers.

NECESSARY DOCUMENTATION

You don't really need any additional documentation to follow the examples in this book. To start writing your own applications, however, an essential item is a reference covering PM API functions with their data types and arguments. You can get this as part of a programmer's kit from IBM or Microsoft, or separately from an independent publisher.

Microsoft also makes available the QuickHelp (QH) utility. This provides an on-line reference to all the OS/2 kernel and PM APIs as well as to other aspects of OS/2 programming.

TYPOGRAPHICAL CONVENTIONS

The names of API functions, like WinCreateWindow, appear frequently in the text. These will be in the normal font, since they're already so long and so distinctive they don't need to be emphasized with italics or boldface. Similarly, system constants, like WA_WARNING, and message names, like WM_DESTROY, which are normally written in all uppercase, will be shown in the normal font. The names of programs, like XCOPY, and files, like SOUND.DEF and BSESUB.H, will also be in all uppercase.

C language keywords, like *if*, *while*, and *switch*, will be in italics to distinguish them from normal English. The names of variables used in C programs, like *hab*, *msg*, and *cbCount*, will also be in italics.

Program listings are printed in a monospaced font.

2

PRESENTATION MANAGER OVERVIEW

This chapter discusses the Presentation Manager in general terms. We'll talk about the advantages it brings to the user and to the programmer, and we'll review the various forms that programs can take under OS/2, so you can see where PM fits in. We'll also discuss the relationship of PM to other systems, such as the OS/2 kernel and Microsoft Windows.

THE ADVANTAGES OF PM FOR THE USER

Ultimately, PM (and the rest of OS/2) will owe its success to providing real benefits to users of personal computers. What are these benefits?

The "Visual Desktop" Metaphor

The central advantage of graphics-based user interfaces such as PM (and Microsoft Windows and the Macintosh) is that you don't need to remember the commands used to operate the system and your applications.

You have probably used the DOS manual many times to look up some detail, like what option you use to make XCOPY copy subdirectories. But retrieving information from a manual is time-consuming, even if you can

find the manual. It's far faster, and less frustrating, if the possible choices are built into the program. A PM version of XCOPY, for example, might list "copy subdirectories" as a menu choice or dialog box item. You would immediately see this was an option, and select it if necessary. Or, better yet, it would present the directory tree of the file system visually and allow you to copy directories by dragging icons with the mouse. In fact, this is just what the PM File Manager utility does. This intuitive visual approach is easy to learn; you can probably figure it out by experimentation.

All applications written for the PM user interface potentially offer this sort of convenience. Coupled with a good context-sensitive help capability, it should be possible for users to operate most applications without reference to a manual. And, as users of the Macintosh and Microsoft Windows know, a visual interface is not only more efficient, it's also more fun.

A Consistent User Interface

In a typical MS-DOS environment you must learn a different set of commands for every application. Writing a document to a file in WordPerfect, for example, involves a completely different set of keystrokes, and even a different conceptual approach, than does writing data to a file in dBASE IV or Lotus 1-2-3. If you switch frequently from one application to another, this inconsistency creates confusion and inefficiency.

PM provides an opportunity to standardize such commands. Because menus and dialog boxes are built into PM, each application will create the same kind of menus and dialog boxes. The "look and feel" will be familiar, whether you're using a spreadsheet or a word processor.

IBM has developed a set of guidelines that attempt to standardize all programs that run on IBM computers, from mainframes to the PS/2. These guidelines are called the *Systems Application Architecture,* or SAA. SAA sets standards for both the user interface (*Common User Access,* or CUA) and the programming interface (*Common Programming Interface,* or CPI).

The PM visual interface largely conforms to the CUA standard, so that (in theory) an application works just the same on a System 370 as it does on a PM-equipped personal computer. If all applications follow the SAA guidelines, there can be significant improvements in both the ease of learning a new application and the ease of use of a familiar one. The programs in this book follow the CUA for the most part, and from time to time we'll point out the approach recommended by IBM for conformance with this standard. Adherence to at least some parts of the CUA standard may become a yardstick by which applications are judged.

Application Integration

In MS-DOS it was difficult for applications to communicate with one another; each was a stand-alone entity. PM features a simple, standardized system for user-controlled communication between applications: the clipboard. The user can copy part or all of a document or graphic to the clipboard, and then hot-key to a different application and retrieve the contents of the clipboard into the new application. Transferring data from a spreadsheet into a letter requires only a few keystrokes. This is a powerful way to enhance your productivity.

PM also provides program-controlled communication between applications, using kernel interprocess communication features such as shared memory. This feature is called *dynamic data exchange,* or DDE.

Multitasking

OS/2 also provides the user with the advantages of multitasking. Multitasking can take several forms. Typically, a PM user will be able to run several applications in different windows on the screen. The applications will run concurrently, and they all will be visible at the same time. The user can type a letter in one window and at the same time monitor the operation of a spreadsheet program doing a long recalculation, or a communications program downloading data, in another window. Utilities such as alarm clocks and phone directories can also be set up in small windows for ready reference.

Multitasking also permits individual applications to run significantly faster if they take advantage of multiple threads and multiple processes to carry out different tasks (such as disk I/O, computation, and keyboard input) at the same time.

All of these forms of multitasking should improve a user's productivity.

Graphics and Color

Since the PM interface is graphics based, any application running under it can take advantage of graphics and color. Graphics provide a more exciting and effective way to communicate data from the application to the user: a graph is more vivid than a table of figures. Also, some applications, such as

computer-aided design (CAD), desktop publishing, and paint and draw programs, inherently require graphics.

ADVANTAGES OF PM FOR THE PROGRAMMER

If PM offered significant benefits to the user, but none at all to the programmer, it would still be worth using as an application platform. But there are also far-reaching advantages to the programmer.

Simplified User-interface Programming

Much has been made of the fact that it takes dozens of lines of code for a PM program to write a phrase (like the venerable "Hello, world") on the screen. This is missing the point. Writing phrases to the screen is not the chief output medium for PM programs. The real question is, how many lines of code does it take for an old-style MS-DOS program to create a working pull-down menu? PM makes it (comparatively) simple to create menus, windows, and the other elements of the visual interface. In this way it multiplies your programming power.

Easy-to-use Graphics Interface

PM also provides a powerful set of graphics functions. Some MS-DOS C compilers (Microsoft and Turbo C, for example) now include graphics libraries, but the power of PM graphics is far greater. This is a bonus provided by PM beyond the user interface. Not only can you use the PM graphics functions to create any kind of image on the screen, but you also can save the image in memory or to a disk file and combine images and parts of images in complex and powerful ways.

Graphics Device Independence

In MS-DOS you need to write a separate graphics program for every display adapter: the program that runs in VGA won't run in EGA or Hercules (unless the application includes all the necessary graphics routines, which can make it substantially larger). When new graphics standards are introduced, your application must be rewritten. The multiplicity of graphics standards poses major problems in the MS-DOS world.

In PM these problems vanish. A single version of your application runs on any present (or future) graphics hardware. The operating system—not the application—takes care of translating the instructions in your program to those necessary for output to a particular screen, printer, or other device.

Source Code Compatibility

The source code of applications written to the SAA Common Programming Interface standards should, in theory, be portable from the PM environment to minicomputers and mainframes. This nirvana has not yet come to pass, and may not for some time. But it's nice to know that your application has at least the potential of running on almost any size computer.

API Function Structure

If you've programmed in MS-DOS, you know what a nonintuitive job it can be to access the system functions. You put an arbitrary code number in the AH register, place other values in other registers, depending on the function, and then execute the machine-language instruction INT 21. This is most conveniently handled from assembly language programs and requires consultation with the manual at almost every step.

The OS/2 API functions, and particularly the PM functions, are by contrast far easier to work with. They are readily called from higher-level languages like C. Their names, like WinCreateWindow, give a clear idea of what they do, and their arguments are parameters to a function, rather than values placed in an arbitrary handful of registers.

The MS-DOS API grew up in layers over the years, with, for example, a CP/M file system being overlaid by a UNIX-type file system. Redundancies and inconsistencies abound. The PM API is (at least at the moment) very clean, with a consistent, well-organized, well-documented group of system functions. You will find it a pleasure to use.

Of course, the fact that OS/2 provides a multitasking environment simplifies the creation of TSR-type programs. Instead of using the convoluted process needed to write an MS-DOS TSR program, the PM programmer can simply write a normal program and let the user run it in a separate window.

Object-Oriented Programming

PM uses an object-oriented programming approach. Even though the example programs in this book are written in C, the philosophy behind the

program structure is object oriented. We'll touch on this topic as we go along. The object-oriented approach can increase your productivity through code reusability, incremental software development, and easy software maintenance.

The Development Environment

Finally, PM programmers can take advantage of OS/2's multitasking environment in ways similar to those of other users. You can edit in one session while a long compile takes place in another. You can use one session to keep an instance of your editor open on a header file so you can quickly hot-key to it and look up function argument types, while at the same time you write a source file with another instance of the editor. You can set up different environments in different sessions to customize them for particular purposes.

Everyone uses multitasking differently, but no one who has used it wants to go back to a single-tasking development environment.

PM AND OTHER SYSTEMS

Let's see how PM relates to other systems and to the other parts of OS/2.

PM and Microsoft Windows

OS/2 goes beyond Windows by providing true multitasking, a large memory address space, and networking capability. However, the user interface for PM and the one for Windows are almost identical. Windows users already have most of the skills necessary to operate the Presentation Manager.

There are also many similarities between the Windows and PM programming interfaces. The overall structure of programs is similar, and similar functions are used to carry out corresponding tasks. Unfortunately, the names of the functions and their argument types are different, as are many other details. It is not trivial to port an application from Windows to PM, but it is certainly much simpler than porting one from MS-DOS. Microsoft provides documentation for converting Windows programs to PM, and there are even applications to automate the task.

PM and the Macintosh

The Macintosh is descended from the same source as both PM and Microsoft Windows: the work done in the 1970s at the Xerox Palo Alto Research Center (PARC) by Alan Kay and others. It was here that the "visual desktop" metaphor was first realized, with windows on the screen representing pieces of paper on a desk. The Macintosh was the first widely distributed personal computer to employ this approach.

The visual interfaces used in PM, Windows, and the Macintosh are quite similar. All use menus, resizable movable windows with scroll bars, dialog boxes, and similar features. The Macintosh requires a mouse, while in Windows and PM the mouse is optional. Users will find it easy to make the transition from the Mac to a PM system (certainly far easier than going from a Mac to MS-DOS).

The PM API has many similarities and uses many of the same concepts as the Macintosh QuickDraw library. However, the two systems were designed separately from the ground up, and are completely different in detail. It is fairly complex to port applications between the Macintosh and PM.

PM and the OS/2 Kernel

The OS/2 kernel was originally conceived as a stand-alone platform. It was thought that it would have an independent existence as a character-based multitasking operating system, and it was designed so that a variety of different user interfaces—of which PM was one—could be grafted onto it later. Potential customers, however, showed little interest in a text-based version of OS/2. Everyone wants PM, so OS/2 and PM have grown closer together, both from a marketing and a programming standpoint.

Although they serve different purposes, kernel and PM functions are typically used together in an application. Kernel functions provide multithread and multiprocess capability, disk access, memory management, and other fundamental services. The API functions that perform these activities start with the letters "Dos", as in DosRead and DosCreateThread.

The kernel also contains its own character-based input and output functions, left over from its days as a stand-alone platform. Functions starting with "Vio", like VioWrtTTY, write to the screen; those starting with "Kbd", like KbdStringIn, read from the keyboard, and those starting with "Mou", like MouGetPtrPos, get input from the mouse. These I/O routines are not usually compatible with PM, so typically only the kernel functions beginning with "Dos" find their way into PM programs.

Since the focus of this book is on PM, we won't use many kernel functions. However, they do come up occasionally. We'll explain what they do as we go along. Many books provide a full description of the OS/2 kernel. For example, see *Peter Norton's Inside OS/2*, by Robert Lafore and Peter Norton (New York: Brady, 1988); *The Waite Group's OS/2 Programmer's Reference*, by Asael Dror (Indianapolis, Ind.: Howard Sams Company, 1988); and *OS/2 Programming: an Introduction*, by Herbert Schildt (Berkeley, Calif.: Osborne/McGraw-Hill, 1988).

TYPES OF OS/2 PROGRAMS

There are five different *modes*, or ways to run programs, in OS/2, so it's easy to get confused. Let's review the possibilities and see where PM programs fit into the picture.

Compatibility Box

First, and least interesting, you can use the OS/2 compatibility box to run old-style MS-DOS programs. This is sometimes called "Dos-Mode" or the "3.x box." Most MS-DOS applications will run in the box, although some that are timing-dependent or are otherwise ill-behaved may not.

In the 80286 version of OS/2, programs in the compatibility box don't multitask: when you switch to another session, the box stops. The 80386 OS/2 remedies this problem and permits multiple MS-DOS programs to run at the same time as protected-mode programs.

OS/2 Full Screen Command Prompt

The next most sophisticated approach is to run text-based kernel programs in a full-screen window. This looks a lot like MS-DOS: you have the same full-size 80-column by 25-row screen.

Programs that run in this mode use all the kernel functions: those starting with "Dos", "Vio", "Kbd", and "Mou". The Vio, Kbd, and Mou functions are used to place output on the screen, read keyboard input, and control the mouse. Programs in this mode are not usually graphics oriented and cannot use the PM user interface. However, they can take advantage of multitasking, interprocess communication, and the other OS/2 kernel features.

You select "OS/2 Full Screen" as an option from the "Group" menu in PM. You can have many full-screen sessions running at the same time. The foreground session will occupy the entire screen, and background sessions will be seen as items in the Task List window and as icons when you hot-key to the PM screen.

The full-screen command prompt mode is very similar to version 1.0 of OS/2, which did not include PM. Most programs written for version 1.0 will run without alteration in this mode. However, a few functions behave a little differently.

OS/2 Window Command Prompt

In this mode an application runs inside its own window on the PM screen. You can move the window around and resize it, as with other windows. You can also scroll it, so you can see the contents of a full 80 × 25 screen, even when the window is much smaller. However, the windowed command prompt does not run PM applications, but character-mode applications like the full-screen command prompt. With some notable exceptions, you can use most of the same Dos, Vio, Kbd, and Mou calls that you can use with the full-screen command prompt.

You start an OS/2 window from the "Group" menu. Multiple instances of it can run in different windows on the PM screen at the same time.

Presentation Manager Programs

These are the applications discussed in this book. They are completely graphics based, and use menus, windows, and other PM features to communicate with the user. PM functions starting with the letters "Win" manipulate elements of the user interface, and those starting with "Gpi" create graphics elements like lines and circles. PM applications can also use the OS/2 kernel functions. They can't use Vio, Kbd, or Mou functions, since PM has its own way of handling the keyboard, screen, and mouse, but most of the Dos kernel calls are available.

Alphanumeric Presentation Space

An *Alphanumeric Presentation Space* (sometimes called "Advanced Video I/O" or AVIO) is not a mode of operation, but a special kind of window available to PM programs. Unlike other PM windows, AVIO is completely character

based; it can't use graphics or proportional fonts. It is easier to create text in AVIO than in a standard graphics window.

An application using AVIO windows can still use graphics-based menus and dialog boxes, and it can mix normal PM graphics windows with AVIO windows. Text is written to AVIO windows using a subset of the Vio functions from the OS/2 kernel.

AVIO may be appropriate for certain specialized text-based applications. We'll mention AVIO in Chapter 11.

II

THE USER INTERFACE

3

GETTING STARTED

In this chapter you will meet your first PM program. Although it has only four lines of executable code, it is nevertheless packed with mysterious new features. We'll examine the data types used in the program, the header files that define these types, and the API functions that do the work. We'll also see how to compile and link the program. We'll discuss the Make file that automates the development process, and the definition file that supplies the linker with information about the program.

THE SOUND.C PROGRAM

Our example doesn't aspire to be a major application: all it does is beep. However, it is a real, working PM program, and it will serve as a vehicle to explain program syntax and development. Here's SOUND.C, the source file listing:

```
/* ----------------------- */
/* SOUND.C - Sound a note. */
/* ----------------------- */

#define INCL_WIN                        /* include PM Windows headers */
#include <os2.h>
```

```
USHORT cdecl main(void);

USHORT cdecl main(void)
   {
   HAB hab;                          /* handle of anchor block */

   hab=WinInitialize(NULL);          /* initialize PM usage */

   WinAlarm(HWND_DESKTOP, WA_NOTE);  /* sound a note */

   WinTerminate(hab);                /* terminate PM usage */

   return 0;
   }
```

This listing doesn't look very much like a normal C source file. You may be able to guess that the WinAlarm function is responsible for generating the beep when the program is executed, but much of the rest of the program probably remains shrouded in mystery. Let's don our Sherlock Holmes deerstalker's hat, take magnifying glass in hand, and see what we can discover about it.

Prototypes

Other than the USHORT data type, the prototype and definition for *main* should be relatively familiar. The ANSI standard legislates prototypes for all functions, so we supply one. Later we'll move the prototype to a header file, where it will be less intrusive. The *cdecl* keyword indicates that *main* uses the C calling conventions, rather than Pascal or FORTRAN. The *void* type means that main takes no arguments. Normally a program returns a 0 to the operating system to indicate that all went well, as our example does. A nonzero value indicates an error.

Data Types and Header Files

Perhaps the strangest aspect of the program is the use of the data types USHORT and HAB. What kind of types are these? Where did the familiar *int* go?

Derived data types are one of the most distinctive aspects of C language OS/2 source code. A full-size PM program uses these types so extensively that, at a quick glance, it looks like another language.

The data types used in OS/2 programs are defined in a group of header files. USHORT and HAB are defined in a file called OS2DEF.H. (Don't worry that this is not the file *#include*d in the program. We'll discuss header files later.) OS2DEF.H contains the lines

```
typedef unsigned short USHORT      /* us */
typedef LHANDLE HAB;               /* hab */
typedef void far *LHANDLE;
```

You can see that USHORT is nothing but *unsigned short,* and HAB is the same as LHANDLE, which is *void far* *.

OS2DEF.H includes definitions for many common data types and keywords. A few representative ones are

Derived Type	**Equivalent to**
FAR	*far*
SHORT	*short*
INT	*int*
LONG	*long*
USHORT	*unsigned short*
ULONG	*unsigned long*
PSHORT	SHORT FAR *
PLONG	LONG FAR *
PUSHORT	USHORT FAR *
PULONG	ULONG FAR *

There are many other type definitions in OS2DEF.H and in other files, but this shows the general idea. Often one derived type is defined in terms of another, as is true with HAB and PLONG, so that figuring out the basic type is a two-step process. This may seem unnecessarily complicated (some might say outrageously convoluted), but there are good reasons for using these derived types.

Why Use Derived Data Types?

First, and probably most important, using derived types means that you don't need to rewrite your source code for future versions of OS/2. Remember that the definition of the C language deliberately does not specify the size of the various data types, such as *int* and *long.* It is the particular implementation—the hardware and the compiler—that determines the size of the types. Thus, an *int* on an 80286 is 16 bits, while on an 80386, which

uses larger hardware registers, it may be 32 bits. Pointers and other more complex variables may also be different sizes on different hardware or for different compilers.

Derived types anticipate changes in OS/2's platform; from the 80286 to the 80386 and 80486, and to future hardware. The parameters for most API functions require a fixed number of bits. Suppose a parameter was 16 bits and was defined as *int*. If *int* changes from 16 to 32 bits as the OS/2 platform changes from 80286 to 80386, you would need to rewrite your source code. However, if a derived type, defined in a header file, is used to declare the parameter, then when your application is ported to a version of OS/2 on a different platform, only the header file needs to be changed. You may still need to recompile, but you won't need to modify the source code.

Thus, the use of derived data types helps to make OS/2 programs transferable from one hardware-compiler platform to another—an important part of the SAA philosophy (discussed in Chapter 2). From another point of view, using derived data types makes it easy to accommodate changes in future versions of OS/2. If, for example, a "handle of anchor block" is changed from *void far ** to *unsigned int*, no changes are required in the program's source code, and only the line

```
typedef LHANDLE HAB;                /* hab */
```

in the OS2DEF.H file has to be changed to

```
typedef SHANDLE HAB;                /* hab */
```

Another reason for the derived types is brevity. PUSHORT doesn't take as much room on the page (or screen) as *unsigned short far **. Many functions take a half dozen or more arguments. The function's prototype, and the documentation for it, can be made dramatically shorter using the derived types. Being shorter, it is easier to read, and, once you get used to it, more comprehensible.

Finally, the derived types often convey more useful information than a standard type would. For instance, HAB stands for "handle of anchor block." It's easier to learn this than to remember that (at least in our version of OS/2) anchor block handles are type *void far **.

In practice the derived types don't present the problem you might think. Mostly it doesn't really matter what the underlying data type is: you learn that WinInitialize returns a variable of type HAB, and that's all you

need to know. Whether it's *void far ** or *long int* won't change how you think about it or use it.

Data Types and Hungarian Notation

PM documentation makes use of another mnemonic convention. Like derived types, this convention is intended to simplify the programmer's life; and, like derived types, it looks very odd before you get used to it.

Hungarian notation takes its name from the birthplace of its inventor, the Microsoft programmer Charles Simonyi. The idea is to preface a variable name with lowercase letters that indicate the data type or usage of the variable. For example, *us* indicates USHORT, so *usParam* is immediately identifiable as a variable of this type.

Actually, two categories of initial letters are used. The *base type* indicates data type. You can also use a *prefix* to indicate the purpose of the variable. If *hwnd* stands for a window handle, *hwndClient* is recognized as a client window handle. Similarly *a* means an array, and *i* means an index into an array.

The main part of the variable name, following the base and prefix, should be chosen to describe the variable as well as possible. It's written with initial uppercase letters in each word and is sometimes called the qualifier.

The following table shows some common prefixes:

Prefix	**Meaning**
id	Identifier of a resource (window, thread, and so on)
p	Pointer
np	Near pointer
off	Offset (from beginning of structure or array)
c	Count (followed by base type of items counted)
a	Array
i	Index into array (followed by base type of array)
h	Handle
sz	Zero-terminated string
hab	Handle for an anchor block
hwnd	Handle for a window
hmq	Handle for a message queue

The following table shows some common base types:

Base	Meaning
b	BYTE
ch	CHAR
us	USHORT
s	SHORT
ul	ULONG
l	LONG
f	BOOL (can be either TRUE or FALSE)

Combining the base type, prefix, and qualifier produces such variable names as *pchColorValue,* a far pointer to type CHAR; *alTable,* an array of type LONG, and *cbDancingShoe,* a variable that specifies a count of BYTEs.

If there is only one variable of a particular type in a program, it's usual to use only the prefix letters to name it, without the qualifier: for example, *id* or *rc.* In the SOUND.C program this convention has been followed for the anchor block handle, *hab.* If there were two anchor block handles, they could be called something like *habThread1* and *habThread2.*

The suggested Hungarian notation for some data types is given as a comment next to the type definition in the header files, as shown above for HAB in the OS2DEF.H file.

Not every variable uses both a base type and a prefix. In fact, Hungarian notation seems at this point to be a developing art, without firmly fixed conventions.

Also, the compiler won't know whether you're using Hungarian notation or not. It's an option to help avoid coding mistakes, so you're free to use whatever variable names you like. However, Hungarian notation is used in the Microsoft documentation, and it serves a useful purpose, so we'll follow the convention in this book. Don't worry about memorizing all of this, you'll get used to it as you go along.

API Functions

API functions form the heart of OS/2 programming. There are three in the SOUND.C program: WinInitialize, WinAlarm, and WinTerminate.

Notice how readable, self-explanatory, and easy to remember the PM API names are. For the most part they use English words, each starting with an uppercase letter. Some abbreviations are used, such as "Win" for window and "Msg" for message. Compare these names with those used in other systems, such as C function names like *ecvt*, *strspn*, and *cabs.*

Let's look at the three API functions used in our sample program and at APIs in general.

WinInitialize and WinTerminate

Every OS/2 program, no matter how humble, that uses PM's facilities must call WinInitialize before issuing any PM APIs. This function causes OS/2 to set up the necessary data structures to support a PM application, and returns an *anchor block handle.*

Actually, every *thread* in an application needs to request its own anchor block handle. (A thread, if you're unfamiliar with the OS/2 kernel, can be thought of as a C function that runs at the same time as the function that called it.) Our examples (at least until the chapter on multitasking) will use only one thread, so only one anchor block is required.

Another reason to use WinInitialize is that the anchor block handle it returns is used as an argument to many other PM functions.

In this book we'll use boxes to summarize the functions we describe in the text. Here's one for WinInitialize:

Initialize Presentation Manager Facilities

```
HAB WinInitialize(Options)
USHORT Options        Must be NULL

Returns: Handle for the thread's anchor block
```

WinInitialize takes a single argument, which specifies the initialization options. In the current release of the PM there are no options, so you must specify a value of NULL. Future versions of OS/2 may make more use of this parameter.

WinTerminate signals PM that the program (or more accurately a particular thread) does not need PM's services any longer, so the system can deallocate the resources assigned to it. It's usually called just before the program terminates.

Terminate Use of Presentation Manager Facilities

```
BOOL WinTerminate(hab)
HAB hab          Handle for thread's anchor block

Returns: TRUE if successful, FALSE if error.
```

The BOOL data type, used as the return type for WinTerminate, is used for true or false (Boolean) values. This function returns TRUE (a nonzero value) if the function was successful, and FALSE (0) if an error occurred. TRUE and FALSE are defined in OS2DEF.H (as 1 and 0) and should be used in PM programs when appropriate.

This same approach is used for most API functions: They return a useful number if successful, and 0 if an error occurred. (Notice that this is different from the OS/2 kernel APIs. In kernel API calls, a 0 return value indicates success; anything else is an error code number.)

Error Handling

A conscientious application would check the return value after calling an API function and take appropriate action if an error value were returned. For brevity we don't show this in our example. However, the next step would be to call an API function to find out more about the error. There are two choices. WinGetLastError returns an error code, whose meaning can be looked up in the PMWIN.H header file. The other approach is to call WinGetErrorInfo. This returns a structure filled with all sorts of useful information about the error, such as a string that describes the error and can be displayed for the user.

Handles

The *hab* variable returned by WinInitialize is an example of a *handle.* The handle is a popular way to refer to various items in the PM. Besides the anchor block handle, there are window handles, message queue handles, presentation space handles, and many others. The value of the handle is an arbitrary number, which is more efficient to refer to than a name or other identifier. Typically, a handle is obtained when an item is created, used by functions that operate on that item, and then discarded by a terminate function, as in the preceding example.

WinAlarm and Constants

As noted, the function WinAlarm causes the speaker to emit a short beep. This function takes two parameters. The first is another kind of handle: it's

a handle for a window, in this case the window consisting of the entire screen, or desktop, indicated by HWND_DESKTOP. We won't be concerned with exactly what windows are until the next chapter.

The second parameter to WinAlarm determines the kind of beep. There are three choices, which can be specified using the constants 0, 1, and 2; each generates a lower tone than the last. However, the actual values of constants such as these are almost never used as function arguments in PM programs. Instead, identifiers defined in a header file are used, as was true with HWND_DESKTOP.

In this case, the file PMWIN.H contains the lines

```
#define WA_WARNING   0
#define WA_NOTE      1
#define WA_ERROR     2
```

Constants defined in header files are written in all uppercase and use underscores. The first few letters often provide a clue to the function they're used with: here WA stands for WinAlarm.

There are of course many advantages to using identifiers rather than numerical constants. They convey more information and are easier to understand than numerical values, and their use helps the listing to be self-documenting. Because they are defined in generally used header files, they will be recognized by other programmers reading your listing. And, as with derived data types, changes in OS/2 that result in a change to a constant value can be accommodated by changing the header file; it's not necessary to change every instance of the value in the source file.

Generate Audible Alarm

```
BOOL WinAlarm(hwndDesktop, Type)
HWND hwndDesktop     Always HWND_DESKTOP
USHORT fsType        Alarm type (WA_NOTE, etc.)

Returns: TRUE if successful, FALSE if error
```

The only data type in the preceding box that we haven't mentioned yet is HWND, which means "handle to a window." We'll have more to say about window handles in the next chapter.

Function Prototypes

As noted, the ANSI standard specifies prototypes for all functions. Also (unlike normal C library functions like *printf*), the OS/2 functions will not even work without prototypes. This is because, among other reasons, the API functions must be declared to be type *far pascal;* the default *int* won't work.

Where are these prototypes? For the functions in our sample program, they're in the PMWIN.H header file. Here's what they look like:

```
HAB APIENTRY WinInitialize(USHORT);
BOOL APIENTRY WinTerminate(HAB hab);
BOOL APIENTRY WinAlarm(HWND hwndDesktop, USHORT fsType);
```

All API functions are type APIENTRY, that is, they require *far* calls, and all use the Pascal calling sequence.

In the normal C calling convention, a function's arguments are pushed onto the stack from right to left when the function is called, and the calling program must remove the parameters from the stack when the function returns. This makes it possible for a function to have a variable number of arguments. In the Pascal calling convention the arguments are pushed from left to right, and the function removes the arguments from the stack before it returns. The Pascal convention is more efficient, at least on machines with 80x86 chips. It's also used in most languages, other than C, so all the OS/2 API functions use it.

Header Files

We've referred frequently to the header files OS2DEF.H and PMWIN.H. However, if you go back and look at the listing of our sample program SOUND.C (which has now receded many pages into our wake), you'll see that it uses two preprocessor directives, and neither is an *#include* for PMWIN.H. How is this file included in our program?

OS/2 header files are arranged in a hierarchy. Files at the top of the hierarchy contain *#include*s for files lower down. Some of these *#include*s may or may not be activated, depending on whether the listing of the source file *#define*s certain constants.

Why all this complexity? It's a result of trying to meet two conflicting goals. On the one hand you want to minimize the size of the header files actually included in your program, so that compilation will go faster. On

the other hand you shouldn't have to remember dozens of header file names and use different arrangements of them for every program.

Hierarchy of Header Files

Figure 3-1 shows the arrangement of the header files.

At the top of the pyramid is OS2.H. This file is normally *#include*d in every program and is very simple. It contains little more than *#include*s for

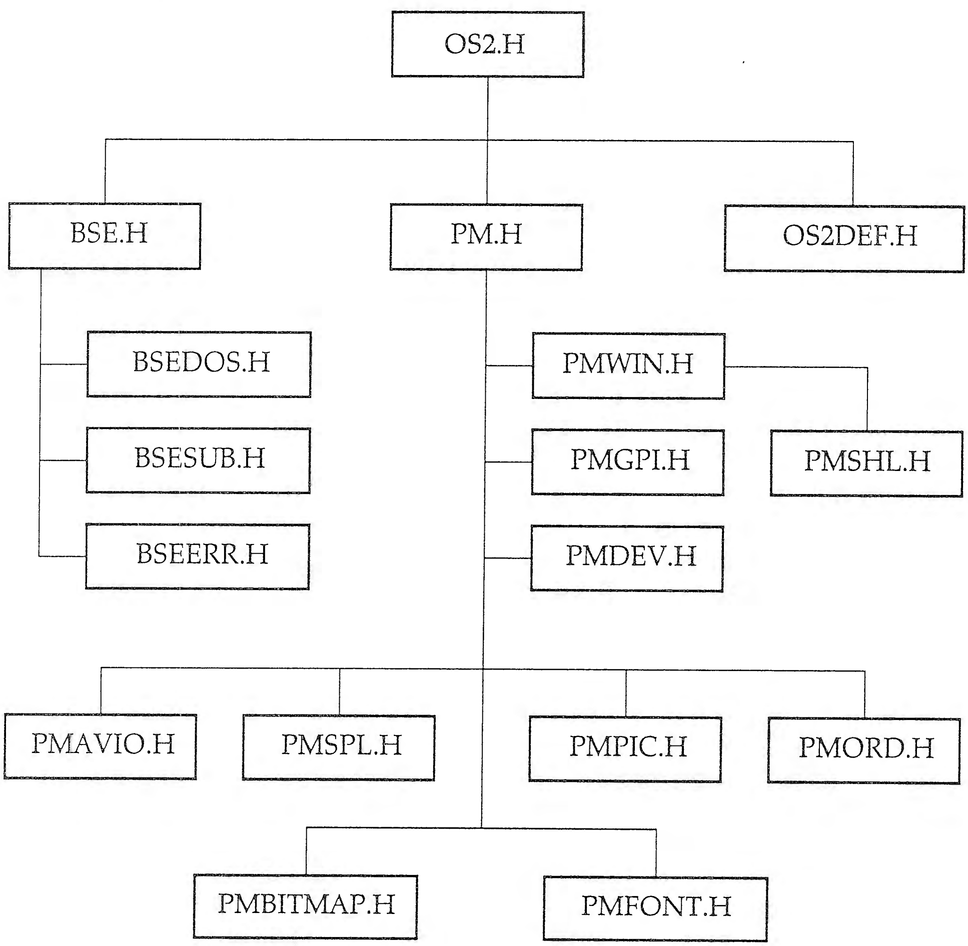


Figure 3-1. Header file hierarchy

three other files: OS2DEF.H, BSE.H, and PM.H.

OS2DEF.H, as we've already seen, contains commonly used definitions.

BSE.H essentially *#include*s three other files: BSEDOS.H, BSESUB.H, and BSEERR.H. The "BSE" in these files stands for "base" and refers to the OS/2 kernel. BSEDOS.H contains prototypes and other data for functions beginning with Dos (like DosSleep), and BSESUB.H covers functions beginning with Vio, Kbd, and Mou (like VioWrtTTY, KbdStringIn, and MouOpen). BSEERR.H contains kernel error message definitions. We won't dwell on these BSExxx.H files, since this is a PM and not a kernel book.

The PM.H file always *#include*s at least three files: PMWIN.H, PMGPI.H, and PMDEV.H. It's not hard to guess that all these files relate to PM functions. PMWIN.H provides prototypes and other data for functions beginning with Win, and PMGPI covers the Gpi functions. PMDEV.H covers a much smaller group of functions that relate to device contexts.

Besides these three files, PM.H may, if certain constants are defined, include various other files such as PMAVIO.H, PMSPL.H, and so on. We won't be concerned with any of these files until much later in the book. The bulk of the functions we'll cover start with Win or Gpi and are prototyped in PMWIN.H and PMGPI.H.

Conditional Compilation

So far you've seen how *#includ*ing OS2.H in our program causes other header files to be dragged in. But the point of the complex header file structure is to minimize compilation time while eliminating unnecessary directives in the source file. How is this accomplished? Conditional compilation eliminates those files, or parts of files, that you don't specifically request.

For instance, we don't use any of the Dos kernel functions in our program. If you look at BSEDOS.H, which contains the prototypes of these functions, you'll see that, in effect, the entire file is bracketed by

```
#ifdef INCL_DOS
```

and *#endif*. So if the example program does not contain a *#define* for INCL_DOS, none of BSESUB.H will be included. The example program does contain the line

```
#define INCL_WIN
```

This causes the entire contents of PMWIN.H to be included. OS2DEF.H is always included along with OS2.H, so we have all the files we need.

There is another level of granularity below the file level. Each file is divided into sections. Each section contains the prototypes for a particular group of functions. For instance, the WinInitialize function is located in the Window Manager section of PMWIN.H. This section is surrounded in effect by

```
#ifdef INCL_WINWINDOWMGR
```

and *#endif.*

If your program contains a *#define* for INCL_WINWINDOWMGR, but not for INCL_WIN, then only the Window Manager section of PMWIN.H will be included with your source file.

You can decide whether you want to include whole header files in your program, or only sections of files. Including only sections may speed your compile times, although for the short examples in this book the effect is not appreciable. We've chosen, for simplicity, to include the whole file PMWIN.H in all our examples.

PROGRAM DEVELOPMENT

Now that you know all about the source file, it's time to learn how to convert it into an actual executable file.

The Make File

Because the compiler and linker both require several switches, and because in more complicated programs there may be other steps besides compiling and linking, it's common to use Make files to automate—and to document—the development of OS/2 programs.

The Make file traditionally uses the same name as the application, but with no extension. Here's SOUND, the Make file for the SOUND.C program:

```
# ---------------
# SOUND make file
# ---------------

sound.obj: sound.c
   cl -c -G2 -W3 -Zp sound.c

sound.exe: sound.obj sound.def
   link /NOD sound,,NUL,os2 slibce,sound
```

If you've never used the MAKE utility, you'll find it's a useful—perhaps even indispensable—addition to your programmer's bag of tricks. You can read all about it in the documentation to your compiler. Briefly, the MAKE utility uses the Make file as instructions for compiling and linking the program.

The Make file contains groups of lines; each group includes two types of statements. The first line in each group describes dependencies. The target file to the left of the colon is dependent on the source files to the right. If one of the source files is dated later than the target file, meaning it has been altered since the target file was last created, then MAKE will invoke the subsequent lines in the group.

The subsequent lines in each group (which must be indented) are the commands executed on the source files to generate the target file.

In the first pair of lines in our example, SOUND.OBJ is recompiled, using CL, whenever SOUND.C is altered. In the second pair, SOUND.EXE is relinked whenever SOUND.OBJ or SOUND.DEF is altered.

When creating your own Make files, you may want to add the Make file itself as a source. For example:

```
sound.obj: sound.c sound
```

This way, if you change the Make file itself (the compiler switches, for example), the target file will be re-created. For simplicity, we did not follow this approach in the book's example programs.

Compiler Switches

Let's look at the command-line switches used by CL.

Because of the complexity of the compile-link process, we don't compile and link in one step. Thus the first switch is -c, which tells CL to compile but not to link. The -G2 switch indicates that 80286 instructions should be used when generating the machine code. Ordinarily, only 8086 instructions are used; adding the additional instructions available in the more advanced processor can result in a more efficient program. If you're compiling for an 80386 and your compiler supports it, you can use a -G3 switch to invoke 80386 instructions.

The -W3 switch specifies the warning level that will be reported by the compiler. There are four levels: 0, 1, 2, and 3. Level 0 means no warning messages at all (although you still get outright errors). Level 1 is the default; levels 2 and 3 report more errors. It's a good idea to use level 3 in

PM programs, since it can tip you off to mismatched data types and other subtle problems. Debugging in PM is hard enough; you might as well take advantage of all the help you can get.

The -Zp switch specifies that structures will be packed on one-byte boundaries. Sometimes structures are packed on word (two-byte or more) boundaries. This is more efficient on some processors, so it is the default for the Microsoft compiler. However, the *#include*d .H files contain structures that describe OS/2 data structures. Since the elements in those structures should not be padded to word boundaries, we need the -Zp switch.

We have not included any of the optimization switches, such as -Ot or -Ox, on the compiler command line. Such optimization would have little effect on our short example programs. Optimization is more appropriately applied by a programmer writing a specific application. Our policy is to avoid optimization switches and the like, unless they are directly applicable to PM programming.

Linker Switches

The compiler normally passes to the linker the name of the library file to be used by the program. This eliminates the need to specify this name on the linker command line. However, the Microsoft C 5.1 compiler currently passes the wrong file name. It passes DOSCALLS.LIB, which was used in version 1.0 of OS/2. The correct name in version 1.2 (and 1.1) is OS2.LIB. So the linker must be told to ignore the library name passed from the compiler. The /NOD (for "no defaults") switch does this.

The rest of the linker command line is fairly straightforward. The object file is SOUND.OBJ, the .EXE file is not specified, since it will have the same name, and we don't want a .MAP file. We're using the small memory model, so the standard protected-mode C library is SLIBCE.LIB. You also need to specify the system library OS2.LIB, since the compiler doesn't pass it to the linker.

The last item in the linker command line tells the linker to find additional information in a file called SOUND.DEF. What's this for?

The Definition File

The definition (.DEF) file provides additional information to the linker, to help it in the creation of the final executable file. Definition files are used to describe dynamic link libraries (DLLs) as well as applications.

The information in a definition file may include, among other items, the stack size, information about the code and data segments in the program, and the names of any routines that must be imported or exported.

Here's SOUND.DEF, the definition file for our example program.

```
; ----------
; SOUND.DEF
; ----------

NAME        SOUND    WINDOWAPI

DESCRIPTION 'Sound a note'

PROTMODE
STACKSIZE   4096
```

The NAME directive specifies that SOUND is an application (rather than a DLL). The WINDOWAPI keyword indicates a PM application. Other options here are WINDOWCOMPAT, for programs that will run in the windowed command prompt, and NOTWINDOWCOMPAT, for programs that require the full-screen command prompt. In this book all our applications are WINDOWAPI. (Unlike the other examples in the book, the SOUND program does not take advantage of any window-related functions and so could actually be defined and run as a non-window application.)

The DESCRIPTION directive is followed by any character string you like, up to 128 characters, surrounded by single quotes. This phrase will be embedded in the header section of the executable file and can be used for copyright notices. PROTMODE specifies a protected-mode program. The alternative is REALMODE, for an MS-DOS program. If neither is specified, the program is assumed to run in both modes. STACKSIZE is marvelously self-descriptive. Our small program doesn't need 4K of stack space, but some OS/2 APIs make use of the caller's stack. To play it safe, Microsoft recommends allocation of 4K of stack space beyond what is used by your program (for local variables and calls).

Creating the Program

To activate the MAKE utility you type

```
make sound
```

Compiling and linking then take place automatically.

The MAKE utility needs to know where the compiler and linker are, so they should be in the current path, along with MAKE.EXE itself. Typically, the Make file and all the other files for an application are kept in their own directory. This way the source files are available to the compiler and linker by virtue of being in the current directory when the Make file is accessed.

For the present example the pathname might be

```
c:\examples\sound
```

If your program doesn't compile and link correctly, there is probably something wrong with the way your development system is set up. You may need to do some tinkering before everything works as it should. For example, make sure the name you gave the combined library during compiler installation, SLIBCE.LIB, is the same as that supplied to the linker in the Make file.

EXECUTING THE PROGRAM

There are three ways to execute our example program. First, you can run it from the full-screen command prompt, by entering its name at the prompt. Since the .DEF file specifies a PM program, the PM graphics screen will briefly appear when the program executes. You can also run it from a windowed command prompt.

Finally, you can run the application directly from the PM desktop. To do this, you must add the program to the list in a Group window. Select "New" from the "Program" menu in this window. Give the program a title; it can be arbitrary, like "Sound Example". Fill in the complete pathname of the program, including the name of the executable file, as in

```
c:examples\sound\sound.exe
```

Click on the Add button and the title will be added to the Group window. To execute the program, double click on the newly installed entry (icon or title). You won't see anything, but you'll hear the beep. Congratulate yourself: you've successfully developed and executed your first PM program.

Note: For this program to make a sound, the "Warning beep" item in the "Options" menu of the Control Panel utility must be checked. This is the default, so there should be no problem unless you changed it.

If you're impatient for visual instead of auditory output, never fear. The examples in the next chapter are all visual.

4

WINDOWS

Until now we've been talking about the nuts and bolts of program development: the files you need to develop a PM program, the switches you use in the compiler command line, and so forth. These are necessary preliminaries. In this chapter we take our first step into the heart of PM programming. We explore several programs that actually put windows on the screen; and, more than that, we begin talking about the philosophy behind PM.

WHAT IS A WINDOW?

The term "window" means different things, depending on whether you are a user or a programmer. In this book we're going to investigate windows from a programmer's viewpoint, but before we do that, we should be clear about what a user thinks of as a window.

User's View of a Window

To the user, a window is a rectangular area of the screen in which information is displayed. This is often (but not always) a *standard window,* which

ordinarily displays information and also comes equipped with various embellishments. Let's examine the Desktop Manager window as an example, since it is easily accessible.

You will see that the Desktop Manager window is a white, rectangular area surrounded by a yellow line. (The colors of these and other window features can be changed using the Control Panel utility.) You can change the size of the window by dragging this yellow line with the mouse. Or you can select "Size" from the System Menu and press the arrow keys. Almost every mouse operation on a standard window can also be performed with the keyboard.

The yellow line is called the *sizing border.* At the top of the window is a colored area with the application's title in the middle. This is the *title bar.* To the left of the title bar is the system menu symbol, and to the right are the minimize and maximize icons. On the right side of the window is a vertical *scroll bar.*

Some of the area of the Desktop Manager window is occupied by the list of program groups currently available, displayed on a white background.

Programmer's View of a Window

For the user, the window is how it looks on the screen. For the programmer, however, its visual appearance is only a small part of the story. To understand how a programmer thinks of windows, we need to back up and look at program architecture.

Windows, Objects, and Messages

Here, in a nutshell, is one of the major ideas in PM programming: *windows are objects.* If you're familiar with object-oriented programming, this will mean something. If you're not, it probably sounds like an unfathomable Zen riddle. In this book we assume you don't know anything about object-oriented programming, but it's almost impossible to discuss PM programming without using some object-oriented concepts and terms. Let's see if we can make this Zen riddle more meaningful.

Object-Oriented and Traditional Programming

In traditional old-style "procedural" programs, a computer program is conceived to be a series of instructions. We're all familiar with this model: we think of our program as doing something, and then going on to the next

part of the program and doing the next thing. This kind of program architecture is concerned with executing lists of instructions.

A PM program, on the other hand, is more naturally conceived of as a number of windows (or objects). These windows may be visible on the screen, but that's not their essence. Really, a window is mostly a *procedure* (or function). A number of these window procedures sit around in memory waiting to be told what to do. For instance, a window procedure might be responsible for maintaining a standard window on the screen. If the user drags the sizing border of the window, the window procedure is notified of this event. The procedure may then choose to redraw the window to the new size.

Besides its procedure, a window also has certain attributes, such as whether it's visible or not. These procedures and attributes completely define the window.

Messages

The window procedure (or window, which is really the same thing) is notified of events (such as the user clicking the mouse button) by *messages* that are passed to it. As its name implies, a message conveys information. If the user tries to resize the window, the message tells the window, "The user dragged your sizing border." The window can then choose to redraw itself to the new size. Other messages tell a window to move itself, maximize or minimize itself, and so on. When it is not processing messages, a window procedure *isn't doing anything.* It's just sitting there waiting for another message.

Notice the change in emphasis from traditional programs. The window procedures in a PM program are not thought of as *following a list of instructions,* they're thought of as *responding to messages.* The difference is shown in Figure 4-1.

This discussion may seem rather abstract, but keep it in the back of your mind as you read on. It's the underlying philosophy behind windows, which we will discuss in this chapter, and messages, which we will talk about in the next chapter.

Everything Is a Window

Oddly enough, various aspects of a standard window, such as scroll bars, the title bar, and so on, are also called *windows.* They don't look like something you would call a window, but from a programming point of view they act like windows. We could call them "objects," which would serve better to differentiate the programming entity (an object) from

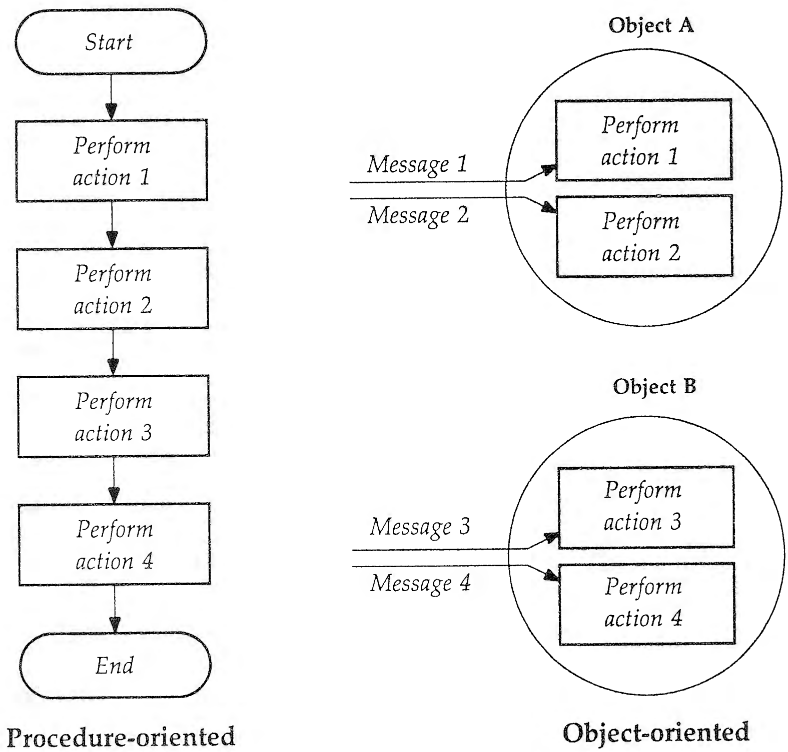


Figure 4-1. Procedural and object-oriented architectures

something we expect to look like a window on the screen. Unfortunately, Microsoft and IBM call them windows, so we will stick to that.

CREATING A VERY SIMPLE WINDOW

Here's a program that puts just a scroll bar on the screen—nothing else. This is not something you ordinarily would do. A program almost always starts with a standard window and places everything else inside this window. For simplicity, however, our scroll bar is drawn on the screen all by itself. Remember that, to a PM programmer, a scroll bar is one kind of window. Here's the listing for SCROLLBR.C:

```
/* ---------------------------------------- */
/* SCROLLBR.C - Create a scroll bar window */
/* ---------------------------------------- */

#define INCL_WIN
#include <os2.h>

USHORT cdecl main(void);

#define WINDOW_ID 1

USHORT cdecl main(void)
   {
   HAB hab;                             /* handle for anchor block */
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* queue message element */
   HWND hwndScrollBar;                  /* handle for scroll bar window */

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);     /* create message queue */

   hwndScrollBar = WinCreateWindow(
                     HWND_DESKTOP,      /* parent */
                     WC_SCROLLBAR,      /* class */
                     NULL,              /* name */
                     WS_VISIBLE | SBS_HORZ,     /* style */
                     100, 100,          /* position */
                     400, 30,           /* size */
                     NULL,              /* owner */
                     HWND_TOP,          /* on top of all siblings */
                     WINDOW_ID,         /* id */
                     NULL,              /* control data */
                     NULL);             /* presentation parameters */

                                        /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwndScrollBar);     /* destroy scroll bar window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */

   return 0;
   }
```

The Make file SCROLLBR and the definition file SCROLLBR.DEF are similar to those files for the example in the previous chapter.

```
# ------------------
# SCROLLBR Make file
# ------------------

scrollbr.obj: scrollbr.c
   cl -c -G2 -W3 -Zp scrollbr.c

scrollbr.exe: scrollbr.obj scrollbr.def
   link /NOD scrollbr,,NUL,os2 slibce,scrollbr
```

```
; -------------
; SCROLLBAR.DEF
; -------------

NAME        SCROLLBR WINDOWAPI
DESCRIPTION 'Scroll bar window'

PROTMODE
STACKSIZE   4096
```

You should run SCROLLBR from the full-screen command prompt. (If you run it from a Group menu, there will be no way to terminate it.) When you run it, you will see a horizontal scroll bar appear on the screen. It looks sort of odd, all by itself: a maverick scroll bar. It will overlay the Desktop Manager and Group Menu windows, or anything else that happens to be in the way. Figure 4-2 shows the output of the SCROLLBR.C program.

Use the mouse to interact with the scroll bar. You can make the arrows at the ends change color (highlight them), you can highlight the *track* (the area between the arrows), and you can drag the *slider* (sometimes called the "thumb" or the "scrollbox") along the track. However, when you press the arrows or click on the track, the slider doesn't move. The scroll bar seems to do some of the things you would expect a scroll bar in a real application to do, but not others. We will explore the reasons for this soon.

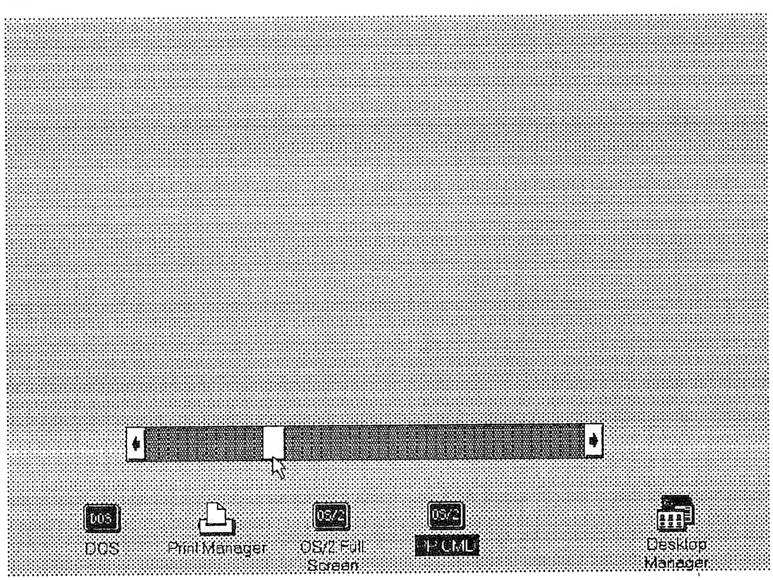


Figure 4-2. Output of SCROLLBAR.C program

Unfortunately, even if you started this program from the full-screen command prompt, there's no way to exit from it gracefully. You will need to click on the OS/2 Full Screen icon, and then choose "Close" from the menu that pops up. This terminates the entire session. (In our next example we will introduce a more elegant exit procedure.)

The WinCreateWindow Function

As you can see from the listing for SCROLLBR.C, the heart of this program is the WinCreateWindow function. This function tells PM that we want to create a scroll bar. Now, does "creating a scroll bar" mean that we simply want to draw a scroll bar shape on the screen? No—there's more to it than that. The scroll bar isn't just a passive shape on the screen, like a box or a pie chart. It *does* things: as you've seen, the slider moves when you drag it with the mouse, and the arrows appear to push in when you click on them.

WinCreateWindow not only draws the scroll bar on the screen, it also arranges for the scroll bar to respond to input from the user. (In fact, in some cases it doesn't draw the scroll bar at all.) What does the scroll bar respond to? That's right: messages. The messages are generated when the user clicks the mouse pointer on the scroll bar and are sent to a procedure that highlights the track, moves the slider, and so on. Where is this procedure? It's in the PM, in a DLL file.

Here's the summary of WinCreateWindow:

Create a Window

```
HWND WinCreateWindow(hwndParent, pszClass, pszName, flStyle, x, y,
                     cx, cy, hwndOwner, hwndInsertBehind, id,
                     pCtlData, pPresParams)
HWND hwndParent                Parent of this window
PSZ pszClass                   Class name
PSZ pszName                    Window text (class specific)
ULONG flStyle                  Window style
SHORT x                        Position of left side of window
SHORT y                        Position of bottom of window
SHORT cx                       Width of window
SHORT cy                       Height of window
HWND hwndOwner                 Window's owner
HWND hwndInsertBehind          Insert just behind this sibling
USHORT id                      Window identifier number
PVOID pCtlData                 Control data (class-specific)
PVOID pPresParams              Presentation parameters (reserved)

Returns: Handle to window, or NULL if error occurred
```

Thirteen arguments is possibly an all-time record. Let's take them one at a time and see how they relate to windows, messages, and other aspects of PM programming.

Genealogy

The first parameter to WinCreateWindow is *hwndParent;* this specifies the parent window of the window we're creating. Every window has a *parent.* This is one of the key concepts in PM programming. The parent-child relationship mostly has to do with how windows are related visually on the screen. Here are the rules:

- When the parent moves, the children move with it.
- The children are drawn on top of the parent.
- Children are clipped at the borders of the parent.

The term *clipped* means that the child can't extend visually beyond the borders of the parent. Any part that extends beyond the parent's border is simply cut off, as a picture is clipped with a pair of scissors to fit a frame.

Children inherit the visibility of their parent. If the parent is visible when the child is created, the child is created visible; if the parent is invisible, its children are created invisible. Also, if the parent is destroyed, the children are destroyed along with it (it does sound biblical).

A parent can have a child, which in turn can have a child. The relationships previously listed apply to this entire family tree. If you move a window, then its children, and its children's children, and indeed all its descendants, move with it. If the relationships of parents and children are unclear to you, try experimenting with an application like the FAMILY.C example at the end of this chapter, or the File Manager utility, which creates different levels of child windows. Notice how the children move when you move the parent, where things are clipped, and what's drawn on top of what.

Normally, some other window would be the parent of a scroll bar. In our example program there is no such parent window, thus the parent of the scroll bar is the entire screen. This is specified using the HWND_DESKTOP constant, whose numerical value is *#define*d in PMWIN.H. The desktop is the first window in the genealogy of any application: all windows are descended from it.

Class

We've talked about ancestry; now we'll talk about class. This may sound like a discussion in anthropology, but in this context *class* refers to categories of windows.

Here's an analogy to make the concept of classes clearer. A normal C variable has a certain data type. For example, the variable *var* might be an *unsigned short* (or USHORT in PM parlance). We would say that the variable *var* is an *instance* of the USHORT data type. In the same way, a particular window in PM is an instance of a window class.

Several window classes are predefined in PM, just as some variable types are predefined in C. These include

WC_SCROLLBAR
WC_TITLEBAR
WC_MENU
WC_BUTTON
WC_FRAME
WC_ENTRYFIELD
WC_LISTBOX
WC_STATIC

WC stands for "Window Class." All these constants are defined to have specific values in PMWIN.H. The window in our example is an instance of the class WC_SCROLLBAR.

You can also define your own window classes; we will introduce this in the next chapter.

Remember that *psz,* the Hungarian prefix for *pszClass,* means a pointer to a zero-terminated string. If you define your own class, you can use a C string for this parameter.

Name

The *pszName* variable is not used in our example. In some windows—a title bar, for example—this name is displayed as part of the window.

Style

The *style* of a window dictates how it will look when it's first created, and some aspects of the way it acts. Some style constants, defined in PMWIN.H, are WS_VISIBLE, WS_MINIMIZED, WS_MAXIMIZED, and WS_SAVEBITS. If we didn't use WS_VISIBLE in our example, the scroll bar would be created invisible, and we would need to make it visible with another API call.

The style constants are defined in PMWIN.H. They can be ORed together to produce combinations of styles. It's best not to assume anything about the value of these constants, or even their size. In our version of OS/2, WS_VISIBLE is a 32-bit number with the value 0x80000000L, but this could change in future versions.

There are two style flags for scroll bars: SBS_HORZ and SBS_VERT. In our example we use SBS_HORZ to determine the orientation of our scroll bar.

Position and Size

The position and size arguments are given in pixels, measured from the lower left corner of the parent window. *Pixels* are the physical dots on the display screen. The dimensions of the screen in pixels are determined by the graphics mode in use. The cached micro presentation space, which we're using now, is by default pixel based, and is thus device dependent. (In Chapter 13 we will discuss using presentation spaces in a device independent way, using coordinates other than pixels.)

Since the parent in our program is the entire screen, the origin of our coordinate system is the lower left corner of the screen. We position the lower left corner of the scroll bar 100 pixels up and 100 pixels right of this origin. The scroll bar is 400 pixels long and 30 pixels wide.

Each window, including the screen itself, has its own coordinate system, starting at its lower left corner, as shown in Figure 4-3.

Ownership

One window can own other windows. This is not the same thing as the parent-child relationship discussed earlier. Ownership has mainly to do with message flow, rather than visual appearance. Ownership is commonly related to control windows, and will be discussed in Chapter 6. Our scroll bar window doesn't have an owner, so this parameter is set to NULL.

Insert Behind

The *hwndInsertBehind* parameter specifies where a window should appear with regard to its siblings. A *sibling* is, as you would guess, another window with the same parent. If our example program had created a second scroll bar, and they were both children of HWND_DESKTOP, they would be siblings. Suppose several siblings overlap on the screen. Which one should be drawn on top of the other? (This is sometimes called *Z-axis positioning*

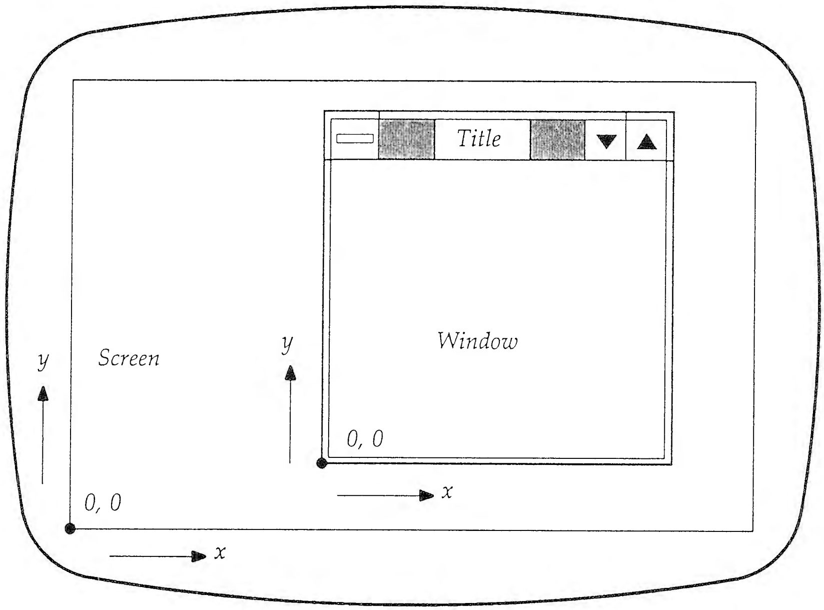


Figure 4-3. PM coordinate system

or *Z-ordering,* where the Z-axis is perpendicular to the plane of the screen.) The *hwndInsertBehind* parameter determines where the window using it will be drawn in relation to its siblings. It can have one of three values: HWND_TOP, HWND_BOTTOM, or the handle of a sibling window. HWND_TOP causes the window using it to be placed on top of all its siblings. HWND_BOTTOM causes it to be placed under all its siblings. Using the handle of a particular sibling causes it to be placed just under that sibling. This fine-tuning is seldom needed, since siblings are usually placed side-by-side so that they don't overlap each other.

Other Arguments to WinCreateWindow

The *id* parameter is used to differentiate one child window from another. The application gives a unique number to each child window. In our example we *#define* WINDOW_ID as 1 at the beginning of the program.

The *pCtlData* parameter is used differently for different classes of windows. For scroll bars it can be used to specify an initial position. We don't use it in our example, so the slider assumes the default position on the far left.

Finally, the *pPresParams* argument can be used to set certain visual attributes of the window, such as color. (These attributes can also be manipulated with WinSetPresParam and related functions.) We don't use this argument in our example, so it is set to NULL.

The WinDestroyWindow Function

Every window created in our application should be destroyed when it's no longer needed. This frees the system resources used by the window. The WinDestroyWindow function is used for this purpose.

Destroy Window

```
BOOL WinDestroyWindow(hwnd)
HWND hwnd        Handle of window to be destroyed

Returns: TRUE if function successful or FALSE if error
```

WinDestroyWindow destroys not only the window whose handle it is given, but all that window's descendants as well.

The Message Loop

There are two new variables in this program: *hmq* and *qmsg*. These variables relate to messages. There are also several new functions that relate to messages: WinCreateMsgQueue, WinGetMsg, WinDispatchMsg, and WinDestroyMsgQueue. Our emphasis in this chapter is on windows rather than messages, so we won't explore the workings of these functions in detail. However, a brief preview may be useful.

The scroll bar we created can receive messages. Some messages, such as those generated when the user clicks the mouse button, are placed in a queue, called the *message queue*. From here they are distributed to the scroll bar window procedure. The WinCreateMsgQueue function creates this queue. To read a message from the queue, our application uses WinGetMsg. To send a message on to the window procedure, it uses WinDispatchMsg. These two functions are arranged in a loop called the *message loop*. Before it terminates, the program uses WinDestroyMsgQueue to deallocate the message queue.

What would happen if you wrote this program without the message processing functions? For one thing, it would not respond to mouse input. The message loop is an essential part of any PM program (at least of any PM program that uses windows).

In the next chapter we will delve more deeply into the workings of the message loop and the functions used in it.

CREATING A STANDARD WINDOW

Our next example creates a standard window. As we noted at the beginning of the chapter, a standard window is what the user thinks of as a window: a rectangular area where information is displayed, with an optional title bar, sizing border, and so on. Here's the listing for STDWIN.C:

```
/* ------------------------------ */
/* STDWIN.C - A standard window.  */
/* ------------------------------ */

#define INCL_WIN
#include <os2.h>

USHORT cdecl main(void);

USHORT cdecl main(void)
   {
   HAB hab;                              /* handle for anchor block */
   HMQ hmq;                              /* handle for message queue */
   QMSG qmsg;                            /* message queue element */
   HWND hwnd;                            /* handle for window */

                                         /* create flags */
   ULONG flCreateFlags= FCF_TITLEBAR ¦ FCF_SYSMENU ¦ FCF_SIZEBORDER ¦
                        FCF_MINMAX ¦ FCF_VERTSCROLL ¦ FCF_HORZSCROLL ¦
                        FCF_SHELLPOSITION ¦ FCF_TASKLIST;

   hab = WinInitialize(NULL);            /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);      /* create message queue */

   hwnd = WinCreateStdWindow(
            HWND_DESKTOP,                /* parent */
            WS_VISIBLE,                  /* style  */
            &flCreateFlags,              /* create flags */
            NULL,                        /* client class */
            " - My first window!",       /* title */
            0L,                          /* client style */
            NULL,                        /* resource module handle */
            0,                           /* resources id */
            NULL);                       /* client handle */
```

```
                                       /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);             /* destroy window */
   WinDestroyMsgQueue(hmq);            /* destroy message queue */
   WinTerminate(hab);                  /* terminate PM usage */

   return 0;
   }
```

The Make and definition files, STDWIN and STDWIN.DEF, are similar to those in the last example.

```
# ----------------
# STDWIN make file
# ----------------

stdwin.obj: stdwin.c
   cl -c -G2 -W3 -Zp stdwin.c

stdwin.exe: stdwin.obj stdwin.def
   link /NOD stdwin,,NUL,os2 slibce,stdwin
```

```
; ----------
; STDWIN.DEF
; ----------

NAME        STDWIN   WINDOWAPI

DESCRIPTION 'Standard window'

PROTMODE
STACKSIZE   4096
```

This program has much in common with the previous example. It has the same message-handling functions and creates a window using one function with many parameters. However, the WinCreateStdWindow function is not quite the same as WinCreateWindow. It is far more powerful.

When you run this program, you will see a standard window on the screen. It has a sizing border, a system menu, a title bar, minimize and maximize icons, and horizontal and vertical scroll bars. The output is shown in Figure 4-4.

You can exercise these features with the mouse (or the keyboard). You can resize the window with the sizing border. You can move it by dragging the title bar. You can maximize it and minimize it with the maximize and minimize windows. And, you can terminate the program by selecting

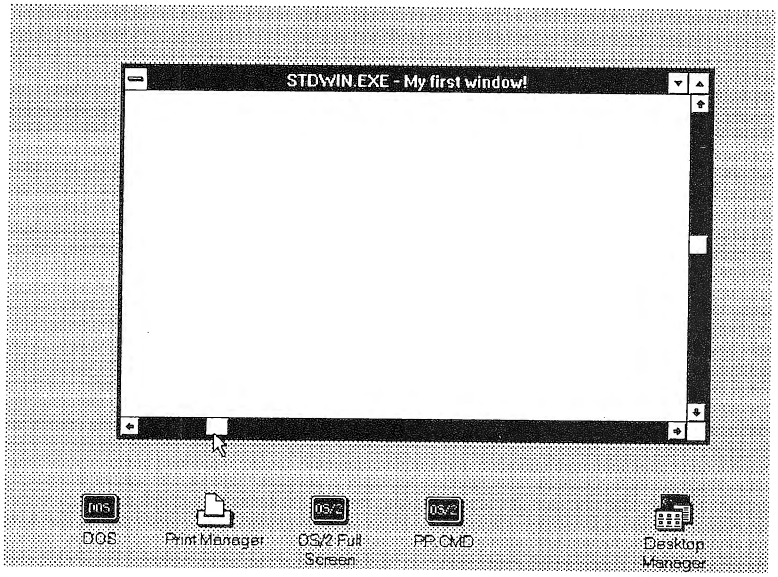


Figure 4-4. Output of STDWIN.C program

"Close" from the System Menu. The scroll bars (like the one in the previous example) do some things, but you can't move the sliders with the arrow icons.

Think how many lines of code it would take to create this standard window in MS-DOS. You'd still be working on it next year. Yet our example program is not that complex: essentially a single function call has given us this entire user interface. We're beginning to see the power built into PM.

The Frame Window

The standard window actually includes another window we haven't mentioned yet: the *frame* window. This window is important: it is the parent of the scroll bar windows, the title bar window, and all the other windows associated with the standard window. The frame window occupies the entire area of the standard window. The system menu, scroll bars, and so on, are its children and are placed on top of it. Figure 4-5 shows the frame window and its children.

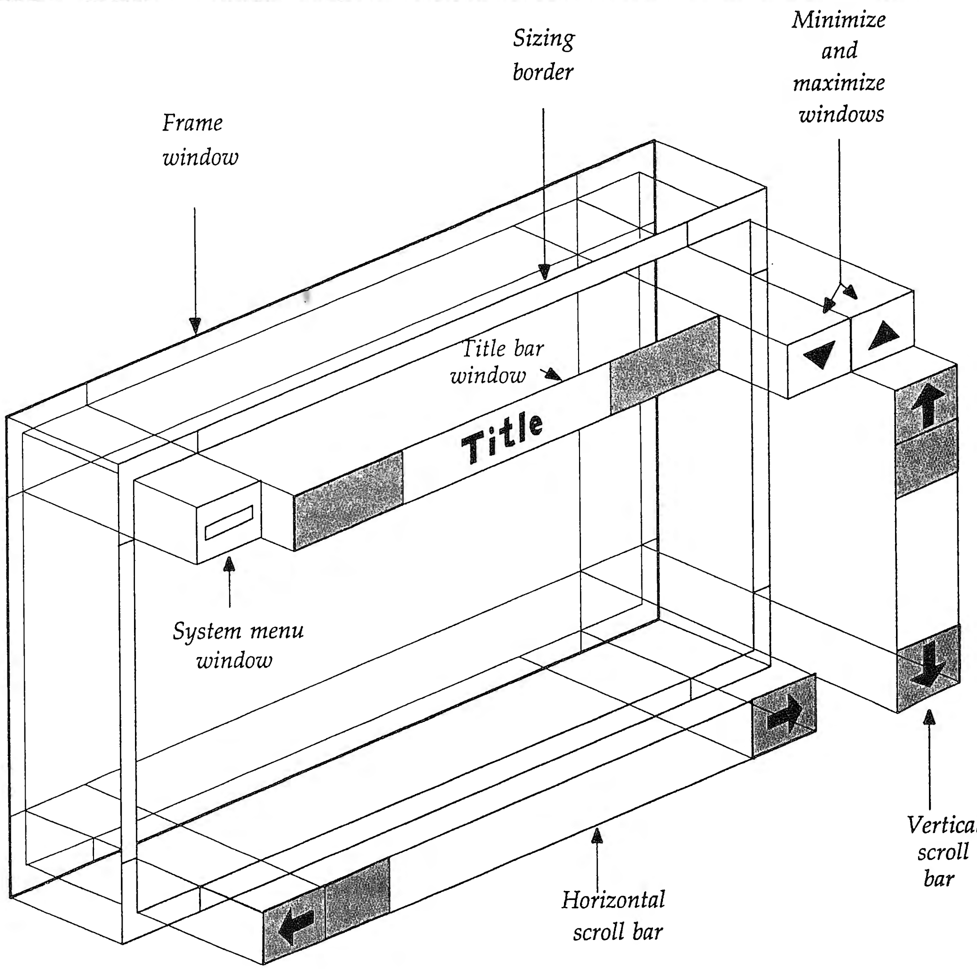


Figure 4-5. The frame window and its children

You may have noticed that the area inside the sizing border, below the title bar, is white (or some other color), not transparent. This is because the frame window has this color. When you call WinCreateStdWindow, it creates the frame window. It then creates the other windows (the title bar, and so on) as children of the frame window. As it happens, the frame window is also the owner of its children, although this fact is not important in this example.

(A minor point to note here is that the sizing border is not really a window: It's a feature drawn directly by the frame window. However, for most purposes it can be thought of as a separate window.)

The WinCreateStdWindow Function

The WinCreateStdWindow function is important in PM programming. It's used at least once in almost every program, and usually several times.

Create a Standard Window

```
HWND  WinCreateStdWindow(hwndParent, flStyle, pflCreateFlags,
                         pszClientClass,  pszTitle, flClientStyle,
                         hmod, idResources, phwndClient)
HWND hwndParent;          Parent window's handle
ULONG flStyle;            Frame window style: WS_ and FS_ constants
PLONG pflCreateFlags;     Child windows included: FCF_MENU, etc.
PSZ pszClientClass;       Class name of client window
PSZ pszTitle;             Title bar text (if FCF_TITLEBAR used)
ULONG flClientStyle;      Client window style
HMODULE hmod;             Resource module handle
USHORT idResources;       Resource ID
PHWND phwndClient;        Client window handle

Returns: Handle of frame window, or NULL if unsuccessful
```

This function has only nine arguments, a comparative luxury after WinCreateWindow. Two of these arguments are used in ways similar to those in WinCreateWindow. First, the *hwndParent* argument specifies the parent of the frame window; in this example it's again HWND_DESKTOP. A window whose parent is the desktop is called a *top-level* (or *main*) window. There can be one or more top-level windows in an application.

Second, the frame style, *flStyle,* is WS_VISIBLE, as it was before. This time, however, it does not include SBS_HORZ, since we're creating a standard window and not a scroll bar.

Create Flags: Child Windows

The create flags argument, *pflCreateFlags,* specifies which windows will be created as children of the frame window. In our example we use the constants FCF_TITLEBAR, FCF_SYSMENU, FCF_SIZEBORDER, FCF_MINMAX, FCF_VERTSCROLL, and FCF_HORZSCROLL. These constants, defined in PMWIN.H, are mostly self-explanatory. They specify that our standard window will have a title bar, system menu, sizing border, minmax windows (minimize and maximize windows together), and vertical

and horizontal scroll bars. You can mix and match these features by ORing the constants together, leaving out any features that aren't appropriate for a particular window. (There's no horizontal scroll bar in the Desktop Manager window, for example.)

How about FCF_SHELLPOSITION? You may have noticed that, unlike WinCreateWindow, WinCreateStdWindow does not include any size or position parameters. The FCF_SHELLPOSITION constant instructs PM to decide for itself where to place the window and how large to make it. If you don't use this constant, the window will be created with zero size, and you will need to use another API to size and locate it.

The final constant, FCF_TASKLIST, informs PM that the program should be placed in the Task List—the list of currently running programs. (You can bring up the Task List window by selecting "Switch to" from a system menu.) Without this constant, the program will not appear on the task list even if it is currently executing.

Another constant, FCF_STANDARD, gives you a group of common FCF_ identifiers already ORed together. We haven't used it here since the constants we need are not the same as those it provides. However, it's useful in some circumstances, and you'll see it used in some PM programming examples.

Client Class, Style, and Handle

One of the windows created as a child of the frame window can be a programmer-defined window. This is called the *client window.* We are not ready to discuss client windows at this point, so all the parameters relating to this window, *flClientStyle, pszClientClass,* and *phwndClient,* are set to zero or NULL.

Resource ID and Module Handle

It's possible to associate a resource with a standard window. Resources are data used by your program. A group of text strings, such as menu items, can be a resource, as can the bitmap of an icon. We won't start using resources until Chapter 7, so the *hmod* and *idResources* arguments are set to NULL and 0, respectively.

Title

The text that appears in the program's title bar typically consists of two diffrent parts, with different origins. The first part is the *program title.* Where does the program title come from? There are two possibilities.

First, let's assume that after you created the program, you installed it in a Group menu. You do this by selecting "New" from the "Program" menu in a particular group and entering the program's title and pathname. Now if you run the program by selecting it from the Group menu, the title you gave it when you added it to the Group menu will appear in the first part of the title bar text. This could be "Stdwin Example" or whatever you entered.

If, on the other hand, you execute the program from the command prompt, the title will be the file name of the program. In the current example this is STDWIN.EXE.

What has been stated so far is true if you have used the FCF_TASKLIST constant in the *pflCreateFlags* argument, as in this example. If you don't use FCF_TASKLIST, the first part of the title bar will be a null ("") string.

The second part of the title bar text consists of whatever you placed in the *pszTitle* argument to WinCreateStdWindow. The SAA guidelines suggest that if your application is processing a data file (say a word processor working on a document file), the name of the data file be used for this second part of the text. A space-dash-space (" - ") combination should separate the program name from the data file name. Thus if your application is a word processor called Word Extra and it's working on a document whose file name is JONES2.TXT, then the title bar text would be "Word Extra - JONES2.TXT". Our example program does not interact with other files, so we use the phrase " - My first window!" instead of a file name. Note that the " - " string is part of this *pszTitle* argument.

In the figures we show the example programs as they look when executed from the command prompt. This is the quicker approach for program development and experimentation.

CREATING MULTIPLE WINDOWS

Our last example in this chapter demonstrates multiple standard windows. It creates two top-level standard windows. It also creates a child for one of these top-level windows: a scroll bar.

Here are the listings of FAMILY.C, the Make file FAMILY, and the definition file FAMILY.DEF.

```
/* ------------------------------- */
/* FAMILY.C - A family of windows. */
/* ------------------------------- */

#define INCL_WIN
#include <os2.h>
```

```
USHORT cdecl main(void);

#define WINDOW_ID 1

USHORT cdecl main(void)
   {
   HAB hab;                              /* handle for anchor block */
   HMQ hmq;                              /* handle for message queue */
   QMSG qmsg;                            /* message queue element */
                                         /* handles for all windows */
   HWND hwndSibling1, hwndSibling2, hwndChildOf2;

                                         /* create flags for sibling 1 */
   ULONG flCreateFlags1 = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                          FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

                                         /* create flags for sibling 2 */
   ULONG flCreateFlags2 = FCF_TITLEBAR | FCF_SIZEBORDER |
                          FCF_MINMAX | FCF_SHELLPOSITION;

   hab = WinInitialize(NULL);            /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);      /* create message queue */

                                         /* create Sibling 1 */
   hwndSibling1 = WinCreateStdWindow(
                    HWND_DESKTOP,        /* top level window */
                    WS_VISIBLE, &flCreateFlags1, NULL,
                    " - Sibling 1", 0L, NULL, 0, NULL);

                                         /* create Sibling 2 */
   hwndSibling2 = WinCreateStdWindow(
                    HWND_DESKTOP,        /* top level window */
                    WS_VISIBLE, &flCreateFlags2, NULL,
                    "Sibling 2", 0L, NULL, 0, NULL);

                                         /* create child of sibling 2 */
   hwndChildOf2 = WinCreateWindow(
                    hwndSibling2,        /* parent */
                    WC_SCROLLBAR, NULL, WS_VISIBLE | SBS_HORZ,
                    100, 100, 400, 30, NULL, HWND_TOP, WINDOW_ID,
                    NULL, NULL);

                                         /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwndChildOf2);       /* destroy child of sibling 2 */
   WinDestroyWindow(hwndSibling2);       /* destroy sibling 2 */
   WinDestroyWindow(hwndSibling1);       /* destroy sibling 1 */

   WinDestroyMsgQueue(hmq);              /* destroy message queue */
   WinTerminate(hab);                    /* terminate PM usage */

   return 0;
   }
```

```
# ----------------
# FAMILY make file
# ----------------

family.obj: family.c
   cl -c -G2 -W3 -Zp family.c

family.exe: family.obj family.def
   link /NOD family,,NUL,os2 slibce,family
```

```
; ----------
; FAMILY.DEF
; ----------

NAME        FAMILY   WINDOWAPI

DESCRIPTION 'A window family'

PROTMODE
STACKSIZE   4096
```

When you run this program, you will see the three windows appear: two standard windows, one with a scroll bar inside. (The scroll bar is the third window.) Figure 4-6 shows the resulting screen display.

The two top-level windows are siblings, since their common parent is HWND_DESKTOP. They are given the titles "Sibling 1" and "Sibling 2", which appear in their title bars. These two siblings are similar but not identical. Sibling 1 uses the FCF_SYSMENU and FCF_TASKLIST constants, while Sibling 2 does not.

Notice how the scroll bar, which is the child of Sibling 2, moves when you move its parent. Also notice that if you move the right-hand or top sizing borders so they cover the scroll bar, the scroll bar will be cut off, or clipped, at that point. The child is automatically clipped at the borders of the parent.

The scroll bar is positioned relative to the lower left corner of its parent (as are all child windows), so if you move the left or bottom sizing borders of Sibling 2, the scroll bar will move, too, keeping the same distance from the borders.

Defining the Family Hierarchy

We've created three windows. Two are top-level standard windows and are siblings. One is a scroll bar, a child of a top-level window. All have window handles, returned from WinCreateStdWindow or WinCreateWindow. The

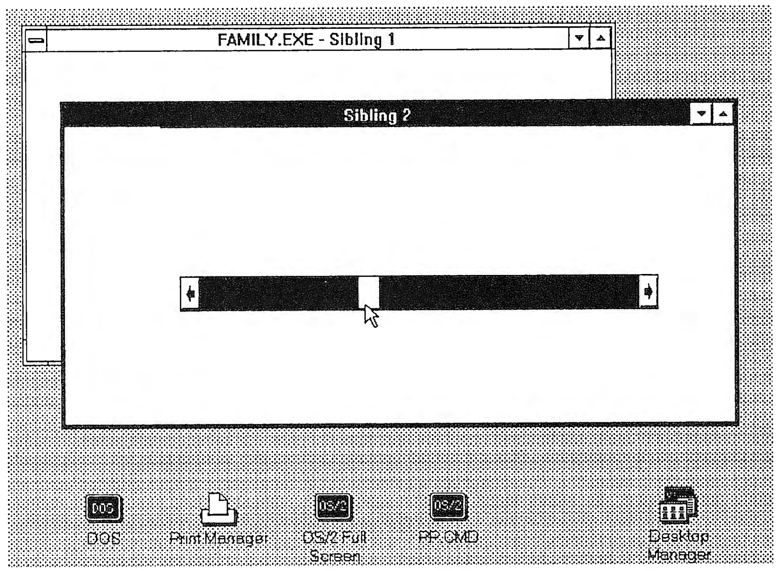


Figure 4-6. Output of the FAMILY.C program

family relationships among these windows is defined using the appropriate handle in the *hwndParent* argument (the first argument) in each call to WinCreateStdWindow or WinCreateWindow. The handle that defines the parent of the two top-level windows is HWND_DESKTOP. The handle that defines the parent of the scroll bar is *hwndSibling2*, the handle of the second top-level window.

By using the appropriate window handle in the *hwndParent* argument each time a window is created, you can construct whatever family relationship you like. A parent can have several children, each of which has its own children, and so on. We'll see more complex examples as we go along.

Making a Window Active

You will notice that whichever sibling window you click on is immediately placed over the other one. Also, its title bar and sizing border change to a distinctive color. If you click on the scroll bar, then Sibling 2 (its parent) rises to the top. What's happening here?

When you click on the title bar of a standard window, its frame window becomes active. Being active is indicated by the standard window (the frame window along with its children) rising to the top of the other

windows on the screen. Additionally, the title bar and the sizing border change color. This happens when you click on the title bar of Sibling 1 or Sibling 2.

When you click on the scroll bar child of Sibling 2, it's Sibling 2 that becomes active. This is because only frame windows can be active. When you click on a window that is not a frame window, the closest ancestor that is a frame window—which could be a parent, grandparent, or more distant ancestor—becomes active.

WINDOWS REVIEW

Let's review what we've learned about windows.

The essence of a window is not how it looks on the screen, but the procedure that defines its operation. The term "window" is misleading, since it implies something visible. In fact, some windows may not be visible at all.

A window receives input in the form of messages and responds appropriately. For example, clicking on the maximize icon in a standard window generates a message to the maximize window procedure. The maximize window procedure then takes the necessary action to expand the window.

Each individual window is an instance of a class, just as an individual orange is a member of the class of oranges. The window classes we have examined in this chapter have been predefined by PM. We've learned about the system menu, title bar, maximize and minimize windows, and scroll bars. Despite their appearance, these are all windows. The window procedures that define these windows are hidden in PM, in a DLL file.

Each window has a parent. The parent-child relationship determines how windows are related visually on the screen. A child window moves with its parent, appears on top of its parent, and is clipped to its parent's boundaries.

A standard window is a term for a collection of windows. The most important window in the standard window collection is called a frame window. The frame window is another predefined window type. The frame window is the parent; some combination of other windows, such as the system menu, title bar, and so on, are the children of the frame window.

In the next chapter we'll focus on messages, and the connection between messages and windows.

5

MESSAGES

In the last chapter we learned that windows are objects—programming entities that take a particular action when they receive information. This information is conveyed in the form of messages. In this chapter, we'll learn more about messages. We'll see what messages really are, and how they embody information. We'll also write our first *window procedures:* the routines in our application that process messages. Along the way we'll pick up some other tricks, such as how to write text to the screen.

Conceptually, messages are fairly simple. However, this simplicity is shrouded by the details and variations of message handling, and perhaps also by preconceived notions based on old-style procedural programs. Thus you may find that your first reading of this chapter leaves unanswered questions. Have faith. The chapters that follow this one, although they focus on other topics, will provide many more examples of messages. As you read them you will absorb the principles of PM's message-based architecture. If you then reread this chapter, you will find that obscure points have clarified themselves, with no conscious effort on your part.

MESSAGES—AN OVERVIEW

Let's begin by briefly discussing messages in a general way, with an eye to casting some light on the examples that follow.

Where Messages Come From

A *message* is information passed to a window. How do messages originate? What entities create them? Since messages are the primary means of communication in the PM environment, they are generated by many different sources.

First, messages can be generated as a direct result of user action. In our SCROLLBR.C program in the previous chapter, a message was generated when the user clicked on one of the scroll bar arrows. The message caused the scroll bar (the scroll bar procedure) to highlight the arrow.

Sometimes PM generates messages as the indirect result of user action. For example, if the user moves a window, and in so doing reveals part of a window underneath, then PM will send a message to the revealed window indicating that it might want to redraw itself.

Messages also can be generated by PM as a result of its own internal processing. For example, when a previously set timer expires, PM sends a message to notify the relevant application of the event.

What Messages Are

In PM programming, a message manifests itself as *a call to a procedure* (function). The values of the arguments passed to the procedure convey the meaning of the message. Notice how well the idea of a message as a call to a procedure fits with the idea of a window (an object) as a procedure. Transmitting a message to a window in effect means making a call to a procedure. Figure 5-1 shows this idea.

Where Messages Go

Where are the window procedures that process messages located? That depends on the type of procedure. There are two broad categories: public and private.

Public window procedures are located in a DLL and are accessible to all applications. A collection of public procedures is predefined by PM. We examined several such public window procedures in the last chapter: title bars, system menus, minimize windows, maximize windows, scroll bars, and the frame window. Since these windows are predefined, we can't examine their code, so it's hard to get a feel for what they are and how they process messages.

Private window procedures, on the other hand, are located within a particular application and are accessible only from within that application. Most of this chapter is devoted to developing example programs that contain private window procedures, and learning how they process messages. In each example program, the private window procedure will define a window called the *client window.* Thus we will call our window procedure the *client window procedure.*

The first half dozen examples in this chapter are variations on a theme. Only part of the program, the client window procedure, changes each time.

DEFAULT MESSAGE PROCESSING

The simplest window procedure needs no message-processing code at all. Instead, it passes all messages on to an API called WinDefWindowProc (for "default window procedure"). This function will take a standard default action that depends on the message. Default message handling is a useful

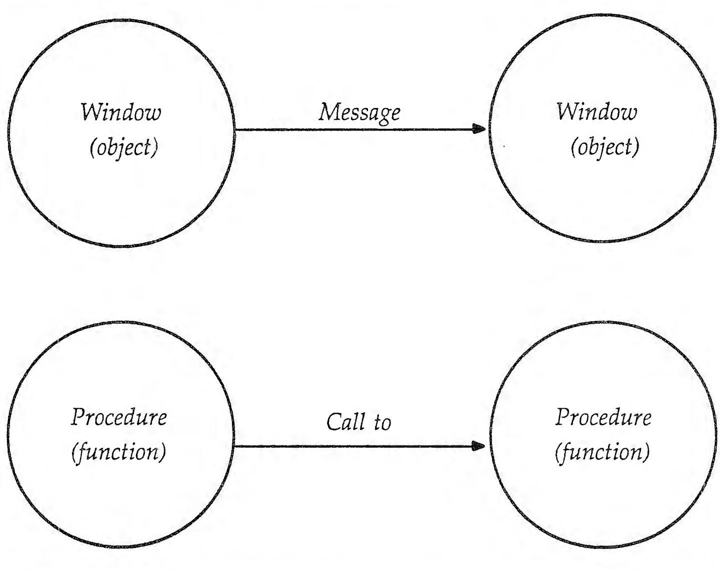


Figure 5-1. Messages and procedure calls

feature of PM: if you don't want to write the code to handle a particular message, you can let the system handle it for you. Since there are dozens of possible messages, this saves a huge amount of complexity and code.

Our first example program installs a client window procedure, but doesn't directly process any messages. It passes all messages on to WinDefWindowProc for default processing. This is the simplest possible client window procedure. The DEFWIN.C listing, with Make file DEFWIN and definition file DEFWIN.DEF, shows how this is done.

```
/* ---------------------------------------------- */
/* DEFWIN.C - Create a standard window with a client */
/* ---------------------------------------------- */

#define INCL_WIN
#include <os2.h>

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

USHORT cdecl main(void)
   {
   HAB hab;                             /* handle for anchor block */
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd;                           /* handle for frame window */

                                        /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";
   HWND hwndClient;                     /* handle for client window */

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab,         /* create message queue */
         0);                            /* of default size */

   WinRegisterClass(                    /* register client window class */
      hab,                              /* hab */
      szClientClass,                    /* name */
      ClientWinProc,                    /* window procedure */
      0L,                               /* style */
      0);                               /* window data */

   hwnd=WinCreateStdWindow(           /* create standard window & client */
           HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
           szClientClass,               /* client class name */
           " - Client Window",
           0L, NULL, 0,
           &hwndClient);                /* handle for client window */

                                        /* messages dispatch loop */
   while (                              /* while not WM_QUIT message */
      WinGetMsg(hab, &qmsg,          /* get a message from thread's queue */
```

```
        NULL, 0, 0))                  /* for any window, any message */
     WinDispatchMsg(hab, &qmsg);     /* dispatch (send) message to window */

   WinDestroyWindow(hwndClient);        /* destroy client window */
   WinDestroyWindow(hwnd);              /* destroy frame window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */

   return 0;
   }

MRESULT EXPENTRY ClientWinProc(         /* client window procedure */
                    HWND hwnd,          /* window handle */
                    USHORT msg,         /* message identifier */
                    MPARAM mp1,         /* message parameter 1 */
                    MPARAM mp2)         /* message parameter 2 */
   {
                                        /* default window processing */
   return(WinDefWindowProc(hwnd, msg, mp1, mp2));
   }
```

```
# ----------------
# DEFWIN make file
# ----------------

defwin.obj: defwin.c
   cl -c -G2s -W3 -Zp defwin.c

defwin.exe: defwin.obj defwin.def
   link /NOD defwin,,NUL,os2 slibce,defwin
```

```
; ----------
; DEFWIN.DEF
; ----------

NAME        DEFWIN WINDOWAPI

DESCRIPTION 'Standard window with client'

PROTMODE
STACKSIZE   4096
```

This program generates a standard window. When you run it, you'll see that it behaves rather strangely: The interior of the window appears to be transparent. Also, it will "capture" whatever is under it. When you drag it to a new location, it carries the underlying image along with it, as a sort of phantom background, as shown in Figure 5-2.

You may think that an opaque white background is normal for a frame window. (If you've changed the window color with the Control Panel, it

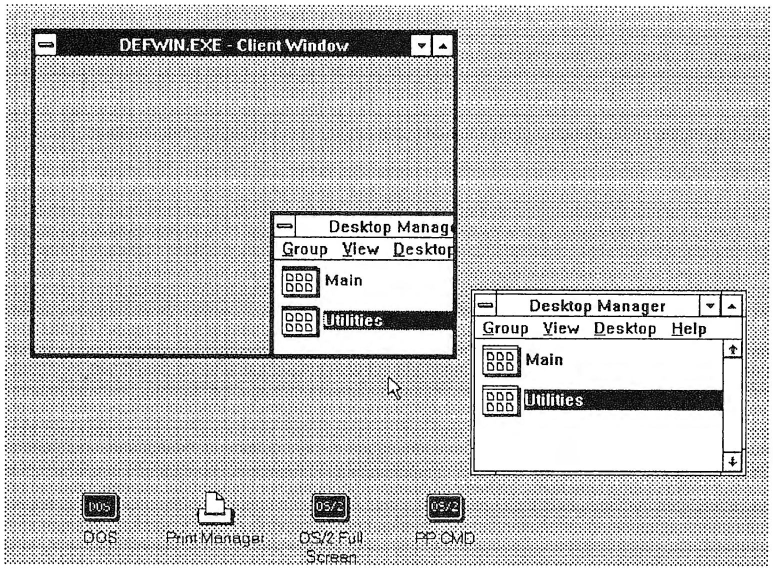


Figure 5-2. Output of the DEFWIN program

may be a color other than white.) That's what we saw with our STDWIN program in the last chapter. Then why is the window in our current example transparent? It turns out that if there is no client window procedure, the frame window will paint its inside white. However, if there *is* a client window procedure, as there is in this example, the frame window won't color itself unless told specifically to do so. We'll see how to color the frame window in the next example. In the meantime, there are plenty of new features to understand about this program.

Program Structure

The DEFWIN.C example contains the two major elements common to all PM programs. These are a *main* procedure and a client window procedure (in our example, a function called ClientWinProc). The *main* procedure will usually do all of the following:

- Initialize PM usage with WinInitialize
- Create a message queue with WinCreateMsgQueue

- Register the client window procedure with WinRegisterClass
- Process messages in a loop with WinGetMsg and WinDispatchMsg
- Create the application's main window with WinCreateStdWindow

When the program is over, the *main* procedure should also remove the window with WinDestroyWindow, destroy the message queue with WinDestroyMsgQueue, and release its access to PM with WinTerminate.

The client window procedure does only one thing: it processes messages. In the current example it does not do this processing itself; instead it forwards all messages to WinDefWindowProc for default processing.

Let's look at *main* and at the client window procedure in more detail.

The *main* Procedure

In previous examples we examined several things *main* does, such as obtaining an anchor block handle and creating a standard window. However, some of the code in the *main* part of DEFWIN.C is new, such as WinRegisterClass, and some, such as the message queue and the message loop, requires more explanation.

Registering the Client Window Class

The windows we've used so far—the scroll bar and the standard window—are members of predefined window classes. We created the scroll bar by using the class identifier WC_SCROLLBAR in the WinCreateWindow function. WinCreateStdWindow automatically creates a frame window plus various other windows that make up the standard window. These windows are all instances of predefined window classes. As we noted, these predefined window classes are *public,* meaning they can be accessed by other applications.

Now we want to create our own window (actually a window procedure). Since all windows are instances of *window classes,* we must start by defining a window class. Defining the class is largely a question of telling PM what procedure is used to process messages for windows in this class. This is the purpose of the WinRegisterClass API. It sets up a correspondence between the name of the class our window will belong to and the procedure that defines that class.

Register a Window Class

```
BOOL WinRegisterClass(hab, pszClassName, pfnWndProc, flStyle,
                      cbWindowData)
HAB hab                  Anchor block
PSZ pszClassName         Name of window class
PFNWP pfnWndProc         Address of window procedure
ULONG flStyle            Class style: CS_MOVENOTIFY, etc.
USHORT cbWindowData      Bytes for per-instance window data

Returns: TRUE if successful, FALSE if error
```

The first argument is the anchor block handle used by our application (actually the particular thread). The second is a string containing the name we give the class. This can be anything we want; after all, we're defining the class. In our example it's "Client Window". The third argument is the address of our client window procedure: *ClientWinProc.* Notice the Hungarian notation *fn* for "function"; *pfn* is the function address. The next argument, *flStyle,* is not used in our example. (It could be one of the CS_ identifiers, as listed in PMWIN.H. Examples are CS_MOVENOTIFY and CS_SIZEREDRAW.)

The last argument, *cbWindowData,* allows you to set aside storage for data used by each instance of the window. We don't make use of this argument in our example.

The key arguments to WinRegisterClass are *pszClassName,* the name of the class we're defining, and *pfnWindowProc,* the address of the procedure that defines the class. A message to any window with this class name will be sent to that procedure for processing.

Creating a Client Window

WinCreateStdWindow creates a frame window. If the proper arguments are included, it also creates a system menu, title bar, and various other control windows, as we saw in the last chapter. It can also create a client window. The client window is (as are the scroll bar, title bar, and so on) a child of the frame window.

If we want WinCreateStdWindow to create the client window, we must give values to several arguments that we filled in with 0 or NULL in previous examples. The fourth argument, *pszClientClass,* is given the name of the client class. In this case it's the variable *szClientClass,* which has the value "Client Window". The sixth argument, *flClientStyle,* could be given

one of the WS_ window styles, but we don't make use of that possibility here. The last argument is the address where PM will return the handle of the client window. This handle will be useful in referring to the window later.

It's taken us three steps to create the client window. First we define the class in ClientWinProc, our client window procedure. This procedure determines how members of the class will react to messages. Second we associate this client window procedure with the name of the class, using WinRegisterClass. And third we create a specific instance of this class of client windows using WinCreateStdWindow.

Destroying the Client Window

In the example we use WinDestroyWindow to return the resources used by the client window to the system, once the program is over. WinDestroyWindow uses the handle to the client window, *hwndClient,* obtained from WinCreateStdWindow. Destroying the client window is not really necessary in this example, since we destroy the parent window, whose handle is *hwnd,* and the children always die with the parent. However, we do it here for clarity, and to show how the handle is used.

The Client Window Procedure

The client window procedure itself is new in this example; we didn't use client window procedures in the last chapter. This procedure looks very much like a normal C function. Its return type is MRESULT EXPENTRY. MRESULT is currently *typedef*ed as VOID FAR * in OS2DEF.H. However, as we've noted before, you shouldn't use the underlying type. In some future version of OS/2, MRESULT might be defined differently, and you would need to rewrite your source code.

EXPENTRY is defined as *pascal far _loadds.* What does the *_loadds* keyword do? A window procedure may be called both from our own application and directly from PM. Since PM sets the DS (data segment register) to its own data, the window procedure needs to reset the DS to its data on entry and restore it on exit. The *_loadds* keyword causes the compiler to add code to do just that, and so it must be used in all window procedure definitions.

In OS/2 1.2 *_loadds* is part of the EXPENTRY *typedef* in the header file, so you don't need to explicitly specify it. However, it does not appear in the older 1.1 header files. Thus, if you're still using OS/2 1.1, you must add

_loadds to your window procedure's function definition, replacing every instance of MRESULT EXPENTRY with MRESULT EXPENTRY *_loadds*.

You should place a prototype for the client window procedure in the main part of the program (or a header file).

Procedure Arguments

There are four arguments to a window procedure, as shown in the following table:

Type	Argument	Use
HWND	*hwnd*	Window handle
USHORT	*msg*	Message identifier number
MPARAM	*mp1*	Message parameter 1 (message specific)
MPARAM	*mp2*	Message parameter 2 (message specific)

These arguments constitute the message and are passed to the window procedure when it is called.

The first argument is the client window handle. Why do we need the window handle? Remember that the client window procedure defines the behavior of all the windows in a particular class. Thus it can be sent messages for any of the windows in the class. The *hwnd* argument tells the procedure which instance of the windows in its class the message is destined for.

The second argument is the message identifier. As with other constants in PM, *#define*d identifiers are used instead of numbers. Identifiers of general window messages mostly start with the characters WM_ (for "Window Message"). There are several dozen of these WM_ messages, like WM_CREATE, WM_PAINT, WM_DESTROY, and so on, defined in PMWIN.H.

The *mp1* and *mp2* arguments contain additional information about the message. They have various meanings, depending on what the message is. We'll start investigating specific messages in the next example, and *mp1* and *mp2* later in the chapter.

Window Procedure Return Values

Once it has received a message, a procedure may want to reply to it: that is, convey some sort of response back to the sender. This is done with the return value of the procedure. The reply is determined by the specific message. If there is no specifically defined reply for a particular message, a return value of NULL (or FALSE) should be used.

The WinDefWindowProc Function

The client window procedure in our example forwards all messages to WinDefWindowProc for default processing. This function takes as arguments the same four variables used as arguments to the window procedure: *hwnd, msg, mp1,* and *mp2.* These variables define the message. The function takes whatever action it thinks makes sense for the message.

WinDefWindowProc returns a default value for each message. Our client window procedure returns this same value, passing it back as its reply to whoever originated the message. Figure 5-3 shows the message flow to WinDefWindowProc.

In future examples we'll see the client window procedure itself processing messages. It can take some or all of the action necessary for a particular message. Even if the client window procedure takes some action on a message, the message can still be passed along to WinDefWindowProc for additional processing.

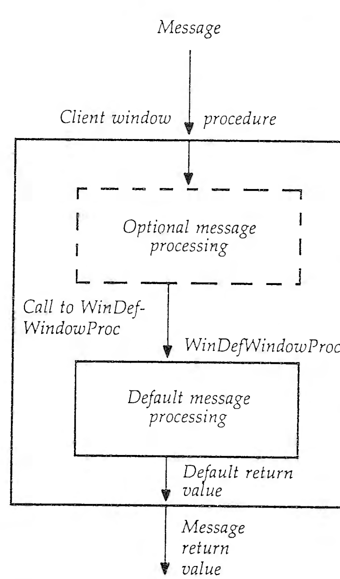


Figure 5-3. Message flow to WinDefWindowProc

The Definition File

Some programmers include an EXPORTS statement for the window procedure in their definition (.DEF) file. This tells the linker that the procedure should be *exported*, or made available outside the application. However, this statement is not needed, since the WinRegisterClass API supplies PM with the address of the window procedure. Using the EXPORTS statement also causes increased overhead, so we don't use it in our examples.

The Make File

There is a subtle change in the Make file for the DEFWIN program. The *-G2* switch in the compiler command line has been changed to *-G2s*. The addition of the "s" (for "stack probes") causes the compiler to disable checking for stack overflow.

There are two reasons for using the *-Gs* switch. If the stack probes are enabled and detect a stack overflow, they send an error message to STDOUT. Writing to STDOUT under PM is not allowed and causes a protection violation. Also, some versions of the Microsoft C compiler perform the stack probes before restoring DS (as instructed by the *_loadds* keyword). This also causes a protection violation. To avoid these problems we use *-G2s* in all our examples.

Some programmers also use the *-Gw* switch, which sets and restores DS for all procedures. However, setting and restoring DS is only needed for window procedures, and this is accomplished by using the *_loadds* keyword, discussed earlier.

THE WM_ERASEBACKGROUND MESSAGE

We've seen what happens when we define a client window class that does no message processing of its own, but forwards all its messages to Win DefWindowProc. The result is a rather unsightly transparent window.

In our next example our client window procedure will take action on a message itself, rather than passing all messages to WinDefWindowProc for default processing. Taking action on this message will eliminate the problem of the transparent window.

The message our example will process is WM_ERASEBACKGROUND. The frame window sends this message to the client window procedure

whenever the frame window needs to be repainted. This happens, for example, when you enlarge the frame window, or uncover part of it. The message flow for WM_ERASEBACKGROUND is shown in Figure 5-4.

Program Variations

Most of the examples in this chapter are variations on the DEFWIN program you've just seen. They use the same *main* procedure, the same Make file, and the same definition file. Only the client window procedure varies from one example to another. (The exception is the MESSAGES.C example at the end of the chapter.) To avoid burdening the chapter with repetitive material, we're going to show only the client window procedures for the next few examples. These examples are called ERASEBKG, CLICK, ACTIVATE, POSITION, and POSTMSG.

If you're typing in the examples, you can copy all the files from DEFWIN to a new directory called (for example) ERASEBKG, change the client window procedure to that shown in the text, and reMAKE the application.

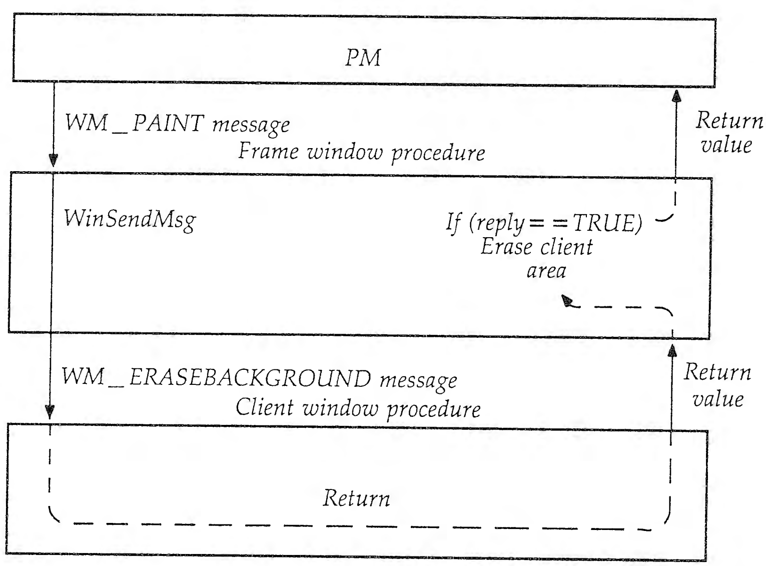


Figure 5-4. Flow of WM_ERASEBACKGROUND message

You don't need to rename files, provided the files for each example are in a separate directory. For example, keep the names DEFWIN.C, DEFWIN, and DEFWIN.DEF for the files in the ERASEBKG directory. That way you don't need to rewrite the Make or definition files.

Here's the listing for the ERASEBKG.C client window procedure:

```
MRESULT EXPENTRY ClientWinProc(        /* client window procedure */
                    HWND hwnd,         /* window handle */
                    USHORT msg,        /* message identifier */
                    MPARAM mp1,        /* message parameter 1 */
                    MPARAM mp2)        /* message parameter 2 */
   {
   switch (msg)
      {
      case WM_ERASEBACKGROUND:
         return TRUE;                  /* yes - erase backgound */
         break;

      default:                         /* default window processing */
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                        /* NULL / FALSE */
   }
```

Instead of always returning the return value of WinDefWindowProc, we now use a *switch* statement to select one of several actions, depending on the message.

Processing WM_ERASEBACKGROUND

The *switch* statement depends on the value of *msg,* the message identifier. If this value is WM_ERASEBACKGROUND, we return TRUE instead of the default value. A return of TRUE as the answer to this message tells the frame window to erase the client area of the frame window. It will fill the window's rectangle with the system window color, which is usually white.

If we try out this new version of the program, we'll see that it behaves much more as we would expect. The client area is no longer transparent, and it no longer "captures" the images under it. It remains white and moves when we move the window. The result is shown in Figure 5-5.

If we had not processed WM_ERASEBACKGROUND ourselves, we would have passed it on to WinDefWindowProc, which would have returned FALSE. When the frame window receives FALSE as an answer to this message, it takes no action. That's why, in our first example, the background remained transparent: The frame window was never told to erase it.

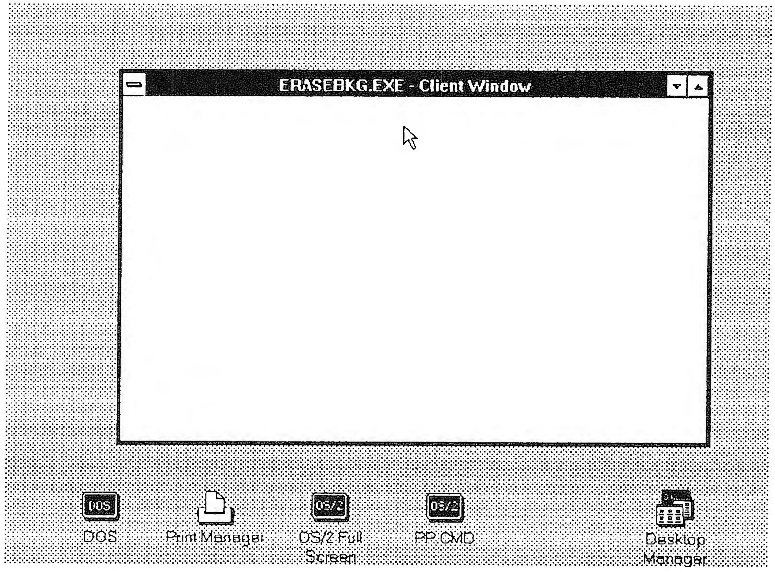


Figure 5-5. Output of ERASEBKG program

Default Processing

In the ERASEBKG example any message other than the the message WM_ERASEBACKGROUND is given the standard default processing: It's sent to WinDefWindowProc, and the return value from this function is returned by our window procedure to whoever sent the message.

MOUSE BUTTON MESSAGES

Let's add to our example the ability to process a different kind of message: one originating from the mouse. Whenever mouse button 1 (which is usually the left button) is pressed, our program will beep. Here's the client window procedure for CLICK.C:

```
MRESULT EXPENTRY ClientWinProc(        /* client window procedure */
                    HWND hwnd,          /* window handle */
                    USHORT msg,         /* message identifier */
                    MPARAM mp1,         /* message parameter 1 */
                    MPARAM mp2)         /* message parameter 2 */
   {
```

```
switch (msg)
   {
   case WM_BUTTON1DOWN:                      /* button 1 clicked */
      WinAlarm(HWND_DESKTOP, WA_NOTE);       /* sound note */
      return TRUE;
      break;

   case WM_ERASEBACKGROUND:
      return TRUE;
      break;

   default:                          /* default window processing */
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }

return NULL;                         /* NULL / FALSE */
}
```

Now there are three possibilities in our *switch* statement. The default and the handling of WM_ERASEBACKGROUND are the same as in previous examples. The new message is WM_BUTTON1DOWN, which PM sends to our application whenever button 1 is clicked while the mouse pointer is in the client window. In response to this message our application sounds a beep with WinAlarm. Following this, we return TRUE to PM to tell PM that we've taken action on the message.

Who Does What?

You may notice an oddity in this program's operation. If you click on another window to make it active, say the Desktop Manager, and then click on the client window of our application (using button 1), the standard window does not become active: the title bar and sizing border remain gray, and the window does not come to the top of the screen—although the beep sounds. Normally when you click on a client window, its top-level standard window becomes active; why not here?

The answer illuminates a significant aspect of what WinDefWindowProc does. Our client window accepts the WM_BUTTON1DOWN message, sounds the alarm, and returns TRUE. Now when WinDefWindowProc handles a mouse button message, it makes the window's top-level window active. However, we don't call this function; we handle the WM_BUTTON1DOWN message ourselves. So no one makes the window active.

The moral is, if you take responsibility for a message, you may need to handle some activities yourself that are normally handled by WinDefWindow-Proc. (Alternatively, you can take some action, and call WinDefWindow-Proc to handle other actions.)

MAKING WINDOWS ACTIVE

Our next example, ACTIVATE.C, adds the capability to make the standard window active when button 1 is clicked in the client window.

```
MRESULT EXPENTRY ClientWinProc(        /* client window procedure */
                    HWND hwnd,         /* window handle */
                    USHORT msg,        /* message identifier */
                    MPARAM mp1,        /* message parameter 1 */
                    MPARAM mp2)        /* message parameter 2 */
   {
   switch (msg)
      {
      case WM_BUTTON1DOWN:             /* button1 clicked */
         WinSetActiveWindow(HWND_DESKTOP, hwnd);   /* activate window */
         WinAlarm(HWND_DESKTOP, WA_NOTE);          /* sound "note" */
         return TRUE;
         break;

      case WM_BUTTON2DOWN:             /* button2 clicked */
         WinAlarm(HWND_DESKTOP, WA_ERROR);         /* sound "error" */
         break;

      case WM_ERASEBACKGROUND:
         return TRUE;
         break;

      default:                         /* default window processing */
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                        /* NULL / FALSE */
   }
```

In this example there are four parts to the *switch* statement. Button 2 now sounds the deeper WA_ERROR tone whenever it's clicked in the client window. This involves processing WM_BUTTON2DOWN, which is handled much like the processing of WM_BUTTON1DOWN, as can be seen in the listing.

The WinSetActiveWindow Function

The function that makes a window active is WinSetActiveWindow.

Make Top-level Window Active

```
BOOL WinSetActiveWindow(hwndDesktop, hwnd)
HWND hwndDesktop          Desktop handle: HWND_DESKTOP
HWND hwnd                 Window handle

Returns: TRUE if successful, FALSE if error
```

This function makes a frame window active. If the *hwnd* argument is set to a window that is not the frame window, then the frame window that is its parent (or more distant ancestor) is made active. The first argument to this function is always HWND_DESKTOP (or the desktop window handle).

MESSAGE PARAMETERS

So far our client window procedure has analyzed only the *msg* argument sent to it. However, we sometimes need information in addition to the message identifier. This is provided by *mp1* and *mp2*, the third and fourth arguments to our window procedure. Our next example, POSITION.C, shows how *mp1* is used to provide additional information.

```
MRESULT EXPENTRY ClientWinProc(          /* client window procedure */
                    HWND hwnd,           /* window handle */
                    USHORT msg,          /* message identifier */
                    MPARAM mp1,          /* message parameter 1 */
                    MPARAM mp2)          /* message parameter 2 */
   {
   switch (msg)
      {
      case WM_BUTTON1DOWN:
         if (SHORT1FROMMP(mp1) < 100)
            {                            /* pointer on left */
            WinSetActiveWindow(HWND_DESKTOP, hwnd);
            WinAlarm(HWND_DESKTOP, WA_NOTE);
            return TRUE;
            }
         else                            /* pointer on right */
            WinAlarm(HWND_DESKTOP, WA_ERROR);
         break;

      case WM_ERASEBACKGROUND:
         return TRUE;
         break;

      default:                           /* default window processing */
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                          /* NULL / FALSE */
   }
```

In this example, mouse button 1 does different things depending on where the mouse pointer is located in the client window. If the pointer is on the far left of the window (within 100 pixels of the left border), pressing

button 1 sounds the WA_NOTE tone and makes the window active. If it is anywhere else in the client window, it sounds the deeper WA_ERROR note, and doesn't make the window active. This division of the window is shown in Figure 5-6.

Contents of Message Parameters

The *mp1* and *mp2* parameters are declared to be of type MPARAM. This is *#define*d in PMWIN.H as a 32-bit variable. However, these parameters are often packed with two (or more) data items, depending on the particular message they are a part of. If you look up WM_BUTTON1DOWN in the documentation, you'll see that *mp1* contains the mouse position at the time of the button click. The low half of *mp1* contains the X position, and the high half contains the Y position. The second message parameter, *mp2*, contains no information and is set to NULL.

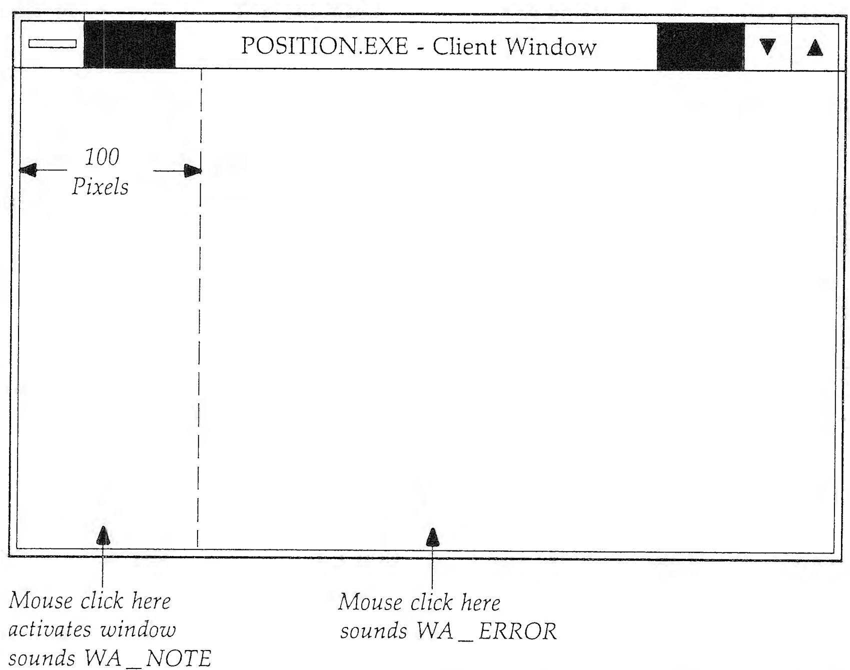


Figure 5-6. Different areas of POSITION window

Using Macros on Message Parameters

You could use the C bit-manipulation operators to extract the high half and low half of message parameters, but you shouldn't. The way data items are packed and even the size of *mp1* and *mp2* may change in future versions of OS/2. To insulate your application from such changes, and to simplify the extraction process, OS/2 provides macros in the PMWIN.H file. There are many such macros. You can extract each of four variables of type CHAR, or each of two SHORTs, or perform other conversions. The macro we want is SHORT1FROMMP, which means "extract the first SHORT from the message parameter." (We could say "low-order" instead of "first," but even the ordering of bytes might be different on different OS/2 platforms.) Its brother, which extracts the second (high order) SHORT, is SHORT2FROMMP. They're currently defined this way in PMWIN.H:

```
#define SHORT1FROMMP(mp)     ((USHORT)(ULONG)(mp))
#define SHORT2FROMMP(mp)     ((USHORT)((ULONG)mp >> 16))
```

We extract the X value with the expression

```
SHORT1FROMMP(mp1)
```

in our client window procedure, and the resulting value of X is used in an *if* statement to determine what the application will do when button 1 is pressed.

MESSAGE FLOW

Before we go on to the next program example, let's look in more detail at how messages travel from one procedure to another. This will lead to an explanation of the message queue and the message loop, which we introduced but did not explain in the last chapter.

We have said that sending a message is similar to calling a procedure. The arguments passed to the procedure, such as *msg*, *mp1*, and *mp2*, convey the meaning of the message. However, this is not the whole story. When a procedure (including PM) wants to send a message to another procedure, it actually has a choice of two methods. It can send the message directly, which is almost the same as making a call to the procedure. Or it can place the message in a queue, from which it is extracted and sent to the recipient procedure later. Let's look at these two approaches in turn.

Sending a Message: Synchronous Transmission

The most direct way for one procedure to send a message to another is to call a function named WinSendMsg. This is almost the same as calling the procedure directly. The arguments sent to WinSendMsg are the same as those received by the procedure: *hwnd, msg, mp1,* and *mp2.*

WinSendMsg calls the recipient and does not return to the caller until the recipient has finished processing the message and returned. Thus the message is sent *synchronously:* the sending process must wait for the completion of the message before it can do anything else. On its return WinSendMsg returns the window procedure's return value.

Posting a Message: Asynchronous Transmission

The second way to transmit a message is to post it to the recipient's message queue. The WinPostMsg function is used for this. In contrast to WinSendMsg, WinPostMsg is *asynchronous.* It returns immediately, so the procedure calling it can go on about its business. The message can be read by the recipient application later and dispatched to the appropriate window. Figure 5-7 diagrams the paths of messages that are sent and posted.

Messages transmitted to a message queue using WinPostMsg are said to be *posted.* Messages transmitted with WinSendMsg are said to be *sent.* (Unfortunately, "send" is such a common word that it is sometimes used generically, even when messages are posted. We will try to avoid this pitfall.)

Sending Versus Posting

Why are two techniques necessary for transmitting messages? An analogy may clarify things.

Sending (as opposed to posting) a message is like making a phone call. You call someone when you have urgent information to impart, or when you want an immediate answer to a question. A procedure might send a message when it needs to be sure an action has been carried out, before it begins another one. For example, PM sends a WM_CREATE message to a new window, telling it to initialize itself. PM then waits for the window to complete processing of this message before transmitting it any other messages.

Posting a message, by contrast, is like mailing a letter. You mail a letter when the message is not urgent, and when you want to drop the letter in the mailbox and go on about other business. PM posts the WM_BUTTON1DOWN when the user presses the mouse button. It posts

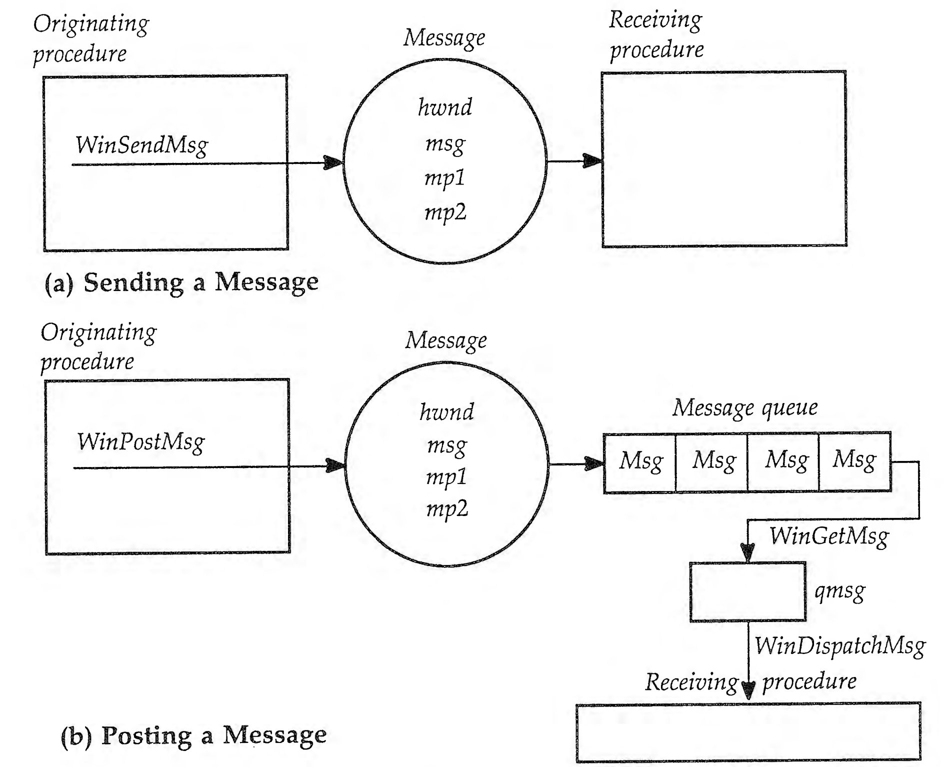


Figure 5-7. Sending versus posting

it because it doesn't require a reply and because the timing is not critical: the window procedure can respond to the message at its leisure, and in the meantime the originator of the message can go on about its business.

Sending a message is fairly straightforward. Posting a message, on the other hand, requires certain supporting features: a message queue to act as a mailbox for incoming messages, and a message loop to take the messages out of the queue and dispatch them to the procedure. Let's examine these features.

The Message Queue

The message queue is created, as we saw earlier, by the WinCreate-MsgQueue API.

Create Message Queue

```
HMQ WinCreateMsgQueue(hab, cmsg)
HAB hab          Anchor block handle
SHORT cmsg       Maximum queue size (0 indicates default size)

Returns: Queue handle, or NULL if error
```

This function must be called after WinInitialize, but before any other window-related PM functions. It creates a message queue for the thread whose anchor block handle is used as the first argument. (A message queue is associated with a particular thread, but you don't need to worry about this in single-thread programs.) The second argument determines the size of the queue and is normally set to 0 to indicate the default queue size. WinCreateMsgQueue returns the handle to the message queue, which is stored in a variable of type HMQ.

When it is no longer needed, the queue is destroyed with WinDestroyMsgQueue.

Destroy Message Queue

```
BOOL WinDestroyMsgQueue(hmq)
HMQ hmq    Message queue handle

Returns: TRUE if successful, FALSE if error
```

We don't need to be concerned with the exact structure or method of storage of messages in the message queue. We do need to know how to extract messages from the queue.

The Message Loop

We've discussed two key elements of the pathway used to process messages. First, we have a function used to process messages: the window procedure. Second, we have a queue where messages can be stored. What determines when a message will be taken from the queue and transferred to the window procedure? This is handled in the message loop. The message loop consists of two functions: WinGetMsg and WinDispatchMsg. The first takes the message from the message queue, and the second sends it (dispatches it) to the window procedure.

You might think these operations should take place in PM, rather than in the application. Or you might think a single function could be used both

to read the message from the queue and to pass it to the window procedure. But putting the process under the control of the application rather than PM, and making it a two-step process, adds important flexibility. For example, the application may read a message from the queue and decide, depending on the message, to pass on a different message, or to not pass it on at all. We'll take advantage of this two-step process in the MESSAGES example at the end of this chapter.

The WinGetMsg Function

The first function in the message loop is WinGetMsg.

Get Message from Message Queue

```
BOOL WinGetMsg(hab, pqmsg, hwndFilter, msgFilterFirst, msgFilterLast)
HAB hab                 Anchor block handle for thread
PQMSG pqmsg             Pointer to QMSG structure
HWND hwndFilter         Window to be filtered (NULL=no filter)
USHORT msgFilterFirst   Beginning of range of messages to be filtered
USHORT msgFilterLast    End of range of messages to be filtered

Returns: FALSE if returned message is WM_QUIT, TRUE otherwise
```

This function reads a message from the thread's message queue and places it in the variable *qmsg*. This variable is a structure of type QMSG, which looks like this (as defined in PMWIN.H):

```
typedef struct _QMSG
        {
        HWND hwnd;    /* Handle of recipient window */
        USHORT msg;   /* Message identifier number */
        MPARAM mp1;   /* Message parameter 1 (message specific) */
        MPARAM mp2;   /* Message parameter 2 (message specific) */
        ULONG time;   /* Time message was posted */
        POINTL ptl;   /* Mouse pointer position when message posted */
        }  QMSG;
```

The first four variables of this structure are the same as those sent to the window procedure. The fifth is the time the message was posted, and the sixth is the position of the mouse pointer at that time.

If no message is available in the queue, WinGetMsg waits; it doesn't return until a message is available.

WinGetMsg returns a value of TRUE for almost all messages. The single exception is WM_QUIT; WinGetMsg returns FALSE when it receives this message. Typically WM_QUIT is generated when the user selects "Close" from the System Menu. Receiving WM_QUIT causes the *while* loop to terminate. The procedure can then execute a termination sequence to release its thread's resources.

Filtration

Ordinarily WinGetMsg reads any message for any window created by the thread. However, it is possible to set up the function to read only some of the messages in the queue. This is known as *filtering.* Filtering is an advanced topic, with certain pitfalls for the unwary, so we won't dwell on it here. Briefly, you can use the *hwndFilter* variable to specify the window to receive messages, and the *msgFilterFirst* and *msgFilterLast* arguments to specify a range of message numbers to be received. Messages to different windows, or outside this range, will be selectively ignored.

When *hwndFilter* is set to NULL, WinGetMsg accepts messages to any window in the queue's thread, and when *msgFilterFirst* and *msgFilterLast* are set to 0, it accepts all messages. This is the usual situation.

The WinDispatchMsg Function

Once it is stored in the QMSG variable, the message must be sent on to its intended recipient procedure. The WinDispatchMsg API accomplishes this.

Send Message to a Window Procedure

```
ULONG WinDispatchMsg(hab, pqmsg)
HAB hab            Anchor block handle
PQMSG pqmsg        Address of QMSG structure containing message

Returns: Value returned by window procedure that it invokes
```

The first argument to WinDispatchMsg is the anchor block handle for the thread. The second is the address of the QMSG structure holding the message.

WinDispatchMsg is similar to WinSendMsg. It performs a sort of remote-control function call. Its effect is to call the window procedure whose handle is specified as the first element of the QMSG structure. However, it doesn't call the procedure directly. Instead it calls PM, which in turn calls the window procedure, using as parameters the first four members of the QMSG structure.

Like WinSendMsg, WinDispatchMsg doesn't return until the window procedure that it called returns. Notice that the format of the arguments: The argument to WinDispatchMsg is the address of the QMSG structure containing the message parameters, while the arguments to WinSendMsg are the message parameters themselves.

To summarize: a message is posted to a queue (of a particular thread). It waits there until read by WinGetMsg, which places it in a variable in the application. From there, it's passed on to the appropriate window procedure by WinDispatchMsg.

Messages and Multitasking

The architecture used by PM for handling messages requires that applications respond quickly to the messages they receive. Why is this important?

Let's distinguish between two kinds of messages. The first kind consists of *user input* messages. These are messages that are caused directly or indirectly by user input through the keyboard or the mouse. WM_CHAR, generated when the user presses a key, and WM_BUTTON1DOWN, generated when the user presses a mouse button, are examples of messages caused by direct user input. The WM_SIZE message, generated when the user changes the size of a window, is an example of a message caused by indirect user input.

The second kind of message is *not* generated as a result of either direct or indirect user input. An example is WM_TIMER, generated when a timer expires.

In the following discussion we'll be concerned with the first kind of message: those relating to user input. Surprisingly, considering the multitasking environment, there is only one user input message active in the system at any given time. Once your application receives such a message, it must finish processing the message before any other message-handling thread, whether in your application or in *any other* application, can receive a user input message. If your application takes too long to respond, all the other applications will be held up waiting for you. It is recommended that after your application returns from WinGetMsg, it takes no longer than 0.1 second before executing WinGetMsg again. Executing WinGetMsg tells PM you've finished processing one message and are waiting for the next one.

Why can't PM handle the processing of several user input messages at the same time? It could, but imagine the consequences. Suppose there are two windows, with A being active and almost covering B. The user clicks on the edge of B to make it active, but before it can become active and move on top, the user tries to select something from a menu in B, which he or she anticipates will appear when B becomes active. Perhaps the system is a little slow in responding, so the user actually ends up clicking on A. Which window should get the mouse message for this click? If multiple active messages were possible, and the mouse message were sent before A

finished processing its current messages, A would get the message, since it's still on top when the click occurs. Logically, however, B should get the message, because the user tried to make it active before using its menu.

To ensure that the system works in this logical way, the process of activating window B must run to completion before the next user input message is accepted. In other words, every such message must be completely processed before any other user input message, anywhere in the system, is acted on.

What happens if your program must handle a lengthy data analysis or disk I/O as a result of the user selecting something from a menu? If your application could not respond to messages until the analysis or I/O (which might take seconds or even minutes) was finished, all the other applications in the system would be frozen for this length of time. This is unsatisfactory for the user. In such cases a second thread should be used to handle the analysis, while the first thread returns immediately to message processing. We'll explore this in Chapter 16, "Multitasking and Objects."

Posting a Message

Our next example, POSTMSG, demonstrates how an application can post a message to itself. Here's the client window procedure. The rest of the example is the same as earlier examples.

```
MRESULT EXPENTRY ClientWinProc(         /* client window procedure */
                    HWND hwnd,          /* window handle */
                    USHORT msg,         /* message identifier */
                    MPARAM mp1,         /* message parameter 1 */
                    MPARAM mp2)         /* message parameter 2 */
   {
   switch (msg)
      {
      case WM_BUTTON1DOWN:                           /* button 1 clicked */
         WinPostMsg(NULL, WM_QUIT, NULL, NULL);    /* post WM_QUIT msg */
         return TRUE;
         break;

      case WM_ERASEBACKGROUND:
         return TRUE;
         break;

      default:                            /* default window processing */
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                          /* NULL / FALSE */
   }
```

When you execute this program, it starts out looking very much like the others we've seen, with a standard window on the screen. However, if you click on the client window, the entire window vanishes, and the program terminates. How does this happen?

Remember that the message loop terminates when WinGetMsg receives a WM_QUIT message. The System menu procedure posts this message to our application when the user selects "Close" from the System menu. However, our application can also post this same message to itself. It does this in response to a WM_BUTTON1DOWN message, by executing WinPostMsg with an argument of WM_QUIT.

Figure 5-8 shows the paths of the messages that result when the user clicks on mouse button 1.

Post a Message to a Message Queue

```
BOOL WinPostMsg(hwnd, msg, mp1, mp2)
HWND hwnd          Handle of window to receive message
USHORT msg         Message number
MPARAM mp1         Message parameter 1 (message specific)
MPARAM mp2         Message parameter 2 (message specific)

Returns: TRUE if message successfully posted, FALSE otherwise
```

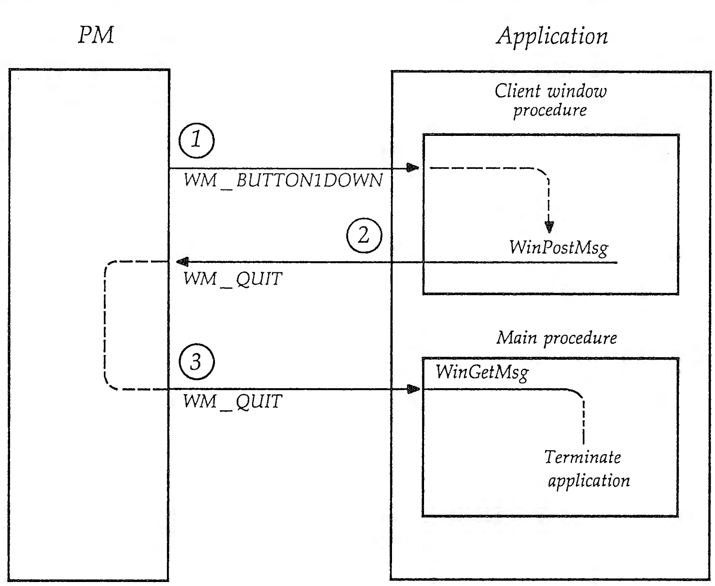


Figure 5-8. Posting a message to yourself

In our example we use a value of NULL as the handle of the window to receive the message. This indicates the message will go into the queue of the current thread.

When WinGetMsg in the message loop receives the WM_QUIT message, it returns FALSE, and the loop and the program terminate, just as if the user had clicked on "Close" in the System Menu.

Passing messages between procedures in the same application, or from a procedure to itself, is an important technique of PM architecture. We'll see other examples as we go along.

Recursion

You should be aware that your client window procedure may be called upon to process two or more messages at the same time. This can happen when a window sends a message to itself. In this case the procedure is called once with the original message and, before it returns, is called again with the message to itself.

You should therefore write window procedures to be *reentrant*. This implies being careful when using external or static variables. If you set the value of such a variable and your window procedure is called recursively to handle a second message before the first call is completed, the variable might be changed by the second call, overriding the value being used by the first call.

Automatic variables pose no problem, since a new set is created each time the procedure is called. Of course, automatic variables don't retain their value when the procedure returns. If you want a variable to retain its value between calls, you can use *per-instance window data*. This requires that the *cbWindowData* argument be set to the size of the per-instance data buffer in WinRegisterClass. Functions like WinSetWindowUShort, WinQueryWindowUShort, and some close cousins are used to insert and read data from this area. We won't pursue this topic here.

THE ROLE OF THE WINDOW PROCEDURE

Now that you've seen some examples of window procedures, we should emphasize two important points about them.

First, most of the real work in a PM program takes place in a window procedure. By "real work" we mean such tasks as statistical analysis, spell-checking a document, or searching for specific data in a disk file—whatever your program is designed to do. You might have formed the impression

that window procedures only handle user interaction, but in fact almost all program activities take place in window procedures, as a result of receiving messages. Thus even large and complex routines will be placed in (or invoked from) the *case* statements for specific messages.

Second, remember that although the term "window procedure" contains the word "window," it doesn't necessarily follow that a window procedure does anything visual. The client window procedures in the examples to date have not drawn anything on the screen. This emphasizes that a window is an object, a procedure that receives messages.

However, a client window procedure can access the screen if necessary. To do so it uses the client window.

WRITING TEXT TO THE SCREEN

Writing to the screen is less important in PM than in traditional programs. In a procedural program the user and the program typically communicate by writing phrases to each other on the screen. In a PM application there are other more effective ways to interact with the user (we'll see some of them in the next chapter, on controls). However, displaying text is a technique we'll need in the final program of this chapter, so let's see how it's done.

Unlike the previous examples in this chapter, this is not a variation of the DEFWIN program. Here are the listings for WMPAINT.C, WMPAINT, and WMPAINT.DEF.

```
/* -------------------------------- */
/* WMPAINT.C - Draw text in window */
/* -------------------------------- */

#define INCL_WIN
#include <os2.h>

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

USHORT cdecl main(void)
   {
   HAB hab;                          /* handle for anchor block */
   HMQ hmq;                          /* handle for message queue */
   QMSG qmsg;                        /* message queue element */
   HWND hwnd;                        /* handle for frame window */

                                     /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";
   HWND hwndClient;                  /* handle for client window */
```

```
    hab = WinInitialize(NULL);          /* initialize PM usage */
    hmq = WinCreateMsgQueue(hab, 0);    /* create message queue */

                                        /* register client window class */
    WinRegisterClass(hab, szClientClass, ClientWinProc,
       CS_SIZEREDRAW,                   /* style = SIZEREDRAW */
       0);
                                       /* create standard window & client */
    hwnd=WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
            szClientClass, " - Client Window", 0L, NULL, 0, &hwndClient);

                                        /* messages dispatch loop */
    while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
       WinDispatchMsg(hab, &qmsg);

    WinDestroyWindow(hwndClient);       /* destroy client window */
    WinDestroyWindow(hwnd);             /* destroy frame window */
    WinDestroyMsgQueue(hmq);            /* destroy message queue */
    WinTerminate(hab);                  /* terminate PM usage */

    return 0;
    }

                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
    {
    HPS hps;
    RECTL rcl;

    switch (msg)
       {
       case WM_PAINT:
          hps = WinBeginPaint(hwnd, NULL, NULL);    /* get PS */
          WinQueryWindowRect (hwnd, &rcl);          /* get win dimensions */
                                                    /* draw text */
          WinDrawText (hps, -1, "text in middle of window", &rcl,
             0L, 0L, DT_CENTER | DT_VCENTER | DT_ERASERECT |
             DT_TEXTATTRS);
          WinEndPaint(hps);                         /* release PS */
          break;

       case WM_ERASEBACKGROUND:
          return TRUE;
          break;

       default:                         /* default window processing */
          return(WinDefWindowProc(hwnd, msg, mp1, mp2));
          break;
       }

    return NULL;                        /* NULL / FALSE */
    }
```

```
# -----------------
# WMPAINT make file
# -----------------
```

```
wmpaint.obj: wmpaint.c
   cl -c -G2s -W3 -Zp wmpaint.c

wmpaint.exe: wmpaint.obj wmpaint.def
   link /NOD wmpaint,,NUL,os2 slibce,wmpaint
```

```
; -----------
; WMPAINT.DEF
; -----------
NAME        WMPAINT WINDOWAPI

DESCRIPTION 'Draw text in window'

PROTMODE
STACKSIZE   4096
```

When you start this application, it writes a phrase in the middle of the client window, as shown in Figure 5-9.

To cause this to happen, the client window procedure has been modified to deal with a new message: WM_PAINT.

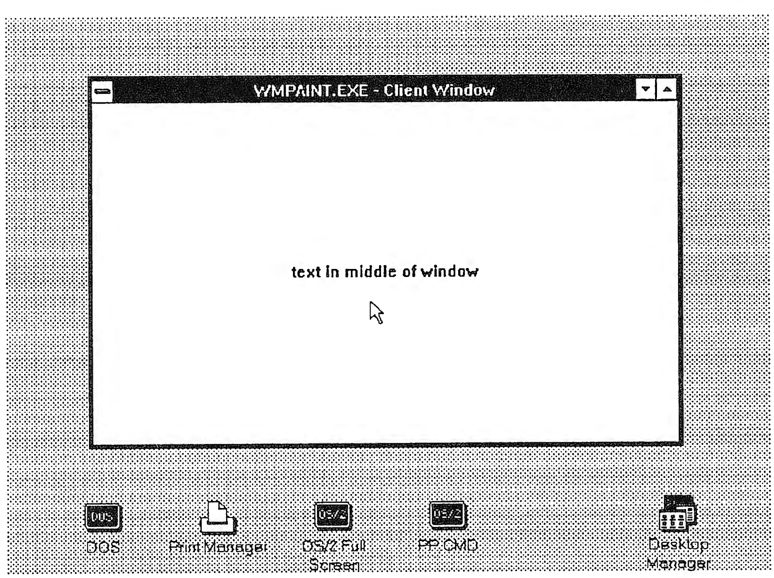


Figure 5-9. Output of WMPAINT Program

The WM_PAINT Message

WM_PAINT is sent to a windows procedure when a part of the window becomes invalid. A region is said to be *invalid* when it does not contain the correct image. For instance, if a window is covered by another window and then uncovered, the uncovered part is invalid. It must be restored to its previous state. Who is responsible for this restoration?

You might think that PM would handle this task. However, sad to say, it does not. PM doesn't really know what an application has placed in a window, so if it were to save the image for later restoration, it would need to save a bit image of the invalid part of the window. Bit images require a lot of storage space, and if PM tried to save the images of many windows at once, it could place an excessive burden on memory. Response time might suffer, too.

PM also won't know what to do if the user enlarges an image. If a window contains text, and the window is enlarged, we don't usually want to enlarge the existing text, we want to write more text to fill up the window. PM doesn't know what text we want to display, so it can't do this. The application itself is in a better position than PM to re-create (or augment) the contents of a window.

Thus an application must be prepared to draw an image not just once, but many times during the course of a program's operation. PM will send a message if the window is enlarged or if it is uncovered by other windows. When it receives this message, the window knows it should repaint (re-draw) itself. The message is WM_PAINT.

WM_PAINT is sent whenever a window needs repainting. It's also sent when a window is first created, so your procedure will know to draw the window contents the first time the window appears. WM_PAINT is *not* normally sent when a window is made smaller. It is also not sent when a window has been moved, since PM normally does take care of redrawing the contents of the window in a new location.

A window procedure should normally draw on the screen only when it receives a WM_PAINT message.

Presentation Spaces

Our client window causes text to be written on the screen. You might be tempted to rephrase this and say that the window "writes to the screen." However, this is not what really happens in PM programming. Instead, the window first writes text (or draws graphics) to an abstract output area

called a *presentation space* (PS). Conceptually, the presentation space is a buffer in memory where an image can be stored. It includes various kinds of data about the image.

From the PS, PM (not your program) takes the text or graphics image and draws it to a physical device, such as the screen. This two-step process makes possible *device independence:* your program doesn't need to worry about the characteristics of displays, display adapters, graphics modes, and similar hardware specifics. All your program has to relate to is the presentation space.

Cached Micro Presentation Spaces

There are several kinds of presentation spaces. The one we use in our example is called a *cached micro PS.* Another important one is called the *normal PS*. The cached micro PS applies only to the screen, not to printers and other graphics devices. PM keeps several of these cached micro PSs available in a cache (storage area), and our application borrows one when it needs it. By contrast, a normal PS must be created before it can be used, which is more complex and time-consuming. A cached micro PS is also somewhat simpler to use than a normal PS. For these reasons we'll use the cached micro PS in the initial chapters of this book.

However, the cached micro PS does not have all the features of the normal PS. The normal PS allows the full use of device independence and has other capabilities as well. We'll discuss presentation spaces in depth in Chapter 10, when we begin our exploration of graphics.

Obtaining a PS with WinBeginPaint

We use a function called WinBeginPaint to obtain a cached micro presentation space.

Begin a Redraw Operation

```
HPS WinBeginPaint (hwnd, hps, prclPaint)
HWND hwnd           Window handle
HPS hps             PS handle, or NULL to obtain cached micro PS
PRECTL prclPaint    Bounding rectangle of invalid region

Returns: Handle to PS if successful, NULL if error
```

The first argument to WinBeginPaint is the handle of the window requesting the PS. If the second argument is NULL, a cached micro PS will be obtained. This is what we want in our example. We don't use the third argument, so it's also set to NULL.

This function returns a handle to a presentation space, which you can then draw on. Note however that drawing on this PS will only update the part of the window that has been made invalid. Other areas of the window cannot be updated with this PS; anything drawn to such areas will not appear. Consequently, WinBeginPaint is only used when processing a WM_PAINT message; that is, when there is an invalid region in the window.

The WinQueryWindowRect Function

Once we've executed WinBeginPaint, we're ready to write to the presentation space. However, we must first find out exactly what area is available for our use. To do this we use WinQueryWindowRect, which tells us the coordinates of our client window. We do this every time we write to the window, since we can never predict how big the window is; the user may have resized it since the last time we wrote to it.

Find Coordinates of Rectangle Bounded by a Window

```
BOOL WinQueryWindowRect(hwnd, prcl)
HWND hwnd           Handle of window whose rectangle is to be found
PRECTL prcl         Address of structure of type RECTL

Returns: TRUE if function successful, FALSE if error
```

The first argument to WinQueryWindowRect is the handle to our client window. The second is the address of the structure where the rectangle's coordinates will be placed. The structure is defined in OS2DEF.H like this:

```
typedef struct _RECTL
        {
        LONG xLeft;
        LONG yBottom;
        LONG xRight;
        LONG yTop;
        } RECTL;
```

Our example provides a structure of this type in the *rectl* variable. Notice the Hungarian notation for the coordinates: x and y are prefixes that indicate horizontal and vertical coordinates. WinQueryWindowRect will fill in this structure with the coordinates of the client window relative to its parent window.

The WinDrawText Function

Finally we're ready to actually draw the text on the screen. This is accomplished with the WinDrawText function. (There are other ways to write to the screen, but this is one of the simplest.)

Draw Line of Text into Rectangle

```
SHORT WinDrawText (hps, cchText, pchText, pcrl, clrFore, clrBack
                   fsCmd)
HPS hps          Presentation space to draw to
SHORT cchText    Length of text string (-1 for ASCIIZ string)
PCH pchText      Address of text string
PRECTL  prcl     Rectangle to contain text
COLOR clrFore    Foreground color (attribute)
COLOR clrBack    Background color (attribute)
USHORT fsCmd     Flags (DT_LEFT, DT_TOP, etc.)

Returns: Number of characters actually drawn
```

The first argument is the handle to the presentation space where drawing will take place. The second is the length of the string to be drawn; if −1 is used, PM will assume an ASCIIZ (0-terminated) string and count characters itself. The third argument is the address of the string, and the fourth is the address of the bounding rectangle to hold the text (in our example we obtained this with WinQueryWindowRect). The next two parameters can be used to specify the foreground and background colors.

The last argument to WinDrawText contains bit flags. These flags, defined in PMWIN.H, can be ORed together. Here are some of the possibilities:

Flag	Meaning
DT_LEFT	Left justify text
DT_RIGHT	Right justify text
DT_CENTER	Center text

Flag	Meaning
DT_VCENTER	Center text vertically
DT_TOP	Text on top
DT_BOTTOM	Text on bottom
DT_WORDBREAK	Break at word end if text doesn't fit
DT_ERASERECT	Erase rectangle before drawing
DT_TEXTATTRS	Use standard colors, ignore *clrFore* and *clrBack*

Our example uses these flags to center the text horizontally and vertically, erase the background before drawing, and apply the PM standard foreground and background text attributes, rather than taking the attributes from *clrFore* and *clrBack* in WinDrawText.

The WinEndPaint Function

When all drawing operations have been completed, WinEndPaint is used to return the presentation space to PM. Issuing WinEndPaint notifies PM that the invalid region of the window has been validated (that is, repainted).

End a Redraw Operation

```
BOOL WinEndPaint(hps)
HPS hps          Handle to PS

Returns: TRUE if successful, FALSE if error
```

The CS_SIZEREDRAW Identifier

The CS_SIZEREDRAW identifier is used in WinRegisterClass to give our client window class the size-redraw style. Whenever a window with this style is resized (made larger or smaller), PM will invalidate the entire window. Without this style, PM will only invalidate a window when it is made larger and will only invalidate the new part of the window. This is unsatisfactory for several reasons.

In our example program we want the text to be displayed in the middle of the window. Thus, when the window has been made smaller, we want to redraw the text. Without CS_SIZEREDRAW the window would not be redrawn and the old text would appear off-center. With CS_SIZEREDRAW, PM will invalidate the entire window whenever it is made

smaller, causing the program to receive a WM_PAINT message and redraw the entire text in the new center of the window.

There is another reason for using CS_SIZEREDRAW. The problem is easier to understand if you imagine that the text being displayed in the middle of the window is only one character long. Assume we don't use CS_SIZEREDRAW. Initially the character is in the middle of the window. Now suppose the user moves the right-hand sizing border just to the left of the character, oblierating it. No part of the window has become invalid, since the window has been made smaller. We don't get a WM_PAINT message, and the character is not redrawn and can't be seen, since its position is now outside the window.

Now suppose that the user moves the right sizing border a little to the right. This creates an invalid region—a vertical strip on the right side of the window. Now the program gets a WM_PAINT message, since there is an invalid region. However, the PS obtained by WinBeginPaint will only update the invalid region. The center of the window, where we want to draw the character, does not fall in this region, and so nothing will be drawn in it. The character remains invisible, even though the window has been enlarged.

Using the size-redraw style solves this problem. It causes the entire window to be invalidated when the window is enlarged, not just the added strip; and it causes WM_PAINT messages to be generated when the window is made smaller as well as larger. With this style the text is redrawn in the center of the window no matter how the user resizes the window.

The Client Window

The client window procedure in our example program obtains a presentation space and draws a text phrase into this space. Since the presentation space is associated with the client window, the image is displayed in the client window. The client window (like the System menu, title bar, and so on) is a child of the frame window. It behaves like other child windows: When you move the frame window, the client window moves with it. The client window lies on top of the frame window, and it is clipped at the frame window's boundaries.

The part of the frame window lying directly under the client window is called the *client area.* Don't confuse this with the client window. Typically, when you write something on the screen, you're writing to the client window. However, when the client window procedure returns a value of TRUE to the frame window in response to WM_ERASEBACKGROUND,

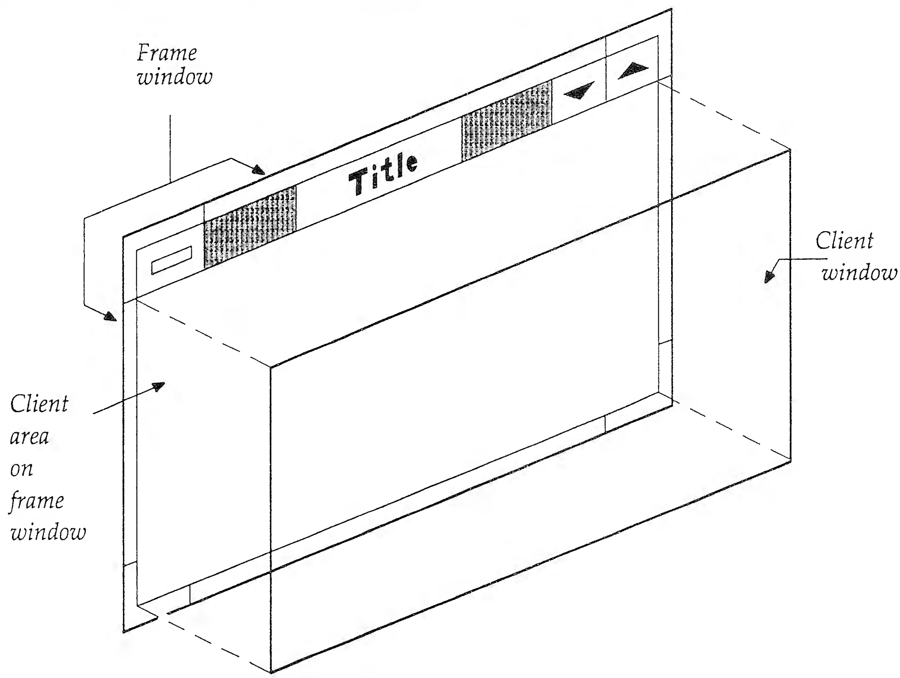


Figure 5-10. The client and frame windows

the frame window erases its client area. Figure 5-10 shows the relationship of the client and frame windows.

THE MESSAGES LEARNING PROGRAM

Messages are a mysterious notion. They move around from window to window, from PM to your application, and you can't see them. To make messages more understandable, we've developed a program that shows you what messages are being received by its client window procedure. Are you curious whether you receive a WM_PAINT message when you move your window? Do you want to know what messages you get when you click on the system menu? Or what happens when you maximize a window? This program will show you.

MESSAGES is not a variation on the DEFWIN program. It uses a header file containing an array that holds all the message names. Here are the MESSAGES.C, MESSAGES, MESSAGES.DEF, and MESSAGES.H files:

```
/* ----------------------------------------------------- */
/* MESSAGES.C - Display messages arriving in client window */
/* ----------------------------------------------------- */

#define INCL_WIN
#define INCL_DOS
#include <os2.h>

#include <stdio.h>                       /* for sprintf() */

#include "messages.h"

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

CHAR *route = "sent";

USHORT cdecl main(void)
   {
   HAB hab;                              /* handle for anchor block */
   HMQ hmq;                              /* handle for message queue */
   QMSG qmsg;                            /* message queue element */
   HWND hwnd;                            /* handle for frame window */

                                         /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Display Messages";
   HWND hwndClient;                      /* handle for client window */

   hab = WinInitialize(NULL);            /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);      /* create message queue */

                                         /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, 0L, 0);

                                        /* create standard window & client */
   hwnd=WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
           szClientClass, "Messages", 0L, NULL, 0, &hwndClient);

                                         /* messages dispatch loop */
   while ( WinGetMsg(hab, &qmsg, NULL, 0, 0))
      {
      route = "posted";
      WinDispatchMsg(hab, &qmsg);
      route = "sent";
      }

   WinDestroyWindow(hwndClient);         /* destroy client window */
   WinDestroyWindow(hwnd);               /* destroy frame window */
   WinDestroyMsgQueue(hmq);              /* destroy message queue */
   WinTerminate(hab);                    /* terminate PM usage */

   return 0;
   }

                                         /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
```

```
   {
   HPS hps;
   RECTL rectl;
   CHAR szText[80];
                                           /* create message name, #, route */
   if (msg > LASTMSG) sprintf(szText, "??? (%#.4x) %s", msg, route);
   else sprintf(szText, "%s (%#.4x) %s", msgList[msg], msg, route);

   hps = WinGetPS(hwnd);                   /* get a PS */
   WinQueryWindowRect (hwnd, &rectl);      /* get window dimensions */
                                           /* display message info. */
   WinDrawText (hps, -1, szText, &rectl, 0L, 0L,
                DT_CENTER | DT_VCENTER | DT_ERASERECT | DT_TEXTATTRS);
   WinReleasePS(hps);                      /* release PS */

   WinAlarm(HWND_DESKTOP, WA_NOTE);        /* sound note */
   DosSleep(500L);                         /* wait 0.5 second */

   return(WinDefWindowProc(hwnd, msg, mp1, mp2));
   }
```

```
# ------------------
# MESSAGES Make file
# ------------------

messages.obj: messages.c messages.h
   cl -c -G2s -W3 -Zp messages.c

messages.exe: messages.obj messages.def
   link /NOD messages,,NUL,os2 slibce,messages
```

```
; ------------
; MESSAGES.DEF
; ------------

NAME        MESSAGES WINDOWAPI

DESCRIPTION 'Display messages arriving at client window'

PROTMODE
STACKSIZE   4096
```

```
/* ---------- */
/* MESSAGES.H */
/* ---------- */

/* WM_* messages */

CHAR *msgList[] =
   {

   /* standard messages */
```

```
"WM_NULL",                  /* 0x0000 */
"WM_CREATE",                /* 0x0001 */
"WM_DESTROY",               /* 0x0002 */
"WM_OTHERWINDOWDESTROYED",  /* 0x0003 */
"WM_ENABLE",                /* 0x0004 */
"WM_SHOW",                  /* 0x0005 */
"WM_MOVE",                  /* 0x0006 */
"WM_SIZE",                  /* 0x0007 */
"WM_ADJUSTWINDOWPOS",       /* 0x0008 */
"WM_CALCVALIDRECTS",        /* 0x0009 */

"WM_SETWINDOWPARAMS",       /* 0x000a */
"WM_QUERYWINDOWPARAMS",     /* 0x000b */
"WM_HITTEST",               /* 0x000c */
"WM_ACTIVATE",              /* 0x000d */
"VOID",                     /* 0x000e */
"WM_SETFOCUS",              /* 0x000f */
"WM_SETSELECTION",          /* 0x0010 */

"WM_PPAINT",                /* 0x0011 */
"WM_PSETFOCUS",             /* 0x0012 */
"WM_PSYSCOLORCHANGE",       /* 0x0013 */
"WM_PSIZE",                 /* 0x0014 */
"WM_PACTIVATE",             /* 0x0015 */
"WM_PCONTROL",              /* 0x0016 */

"???", "???", "???", "???", "???", "???", "???", "???", "???",

"WM_COMMAND",               /* 0x0020 */
"WM_SYSCOMMAND",            /* 0x0021 */
"WM_HELP",                  /* 0x0022 */
"WM_PAINT",                 /* 0x0023 */
"WM_TIMER",                 /* 0x0024 */
"WM_SEM1",                  /* 0x0025 */
"WM_SEM2",                  /* 0x0026 */
"WM_SEM3",                  /* 0x0027 */
"WM_SEM4",                  /* 0x0028 */
"WM_CLOSE",                 /* 0x0029 */
"WM_QUIT",                  /* 0x002a */
"WM_SYSCOLORCHANGE",        /* 0x002b */
"???",                      /* 0x002c */
"WM_SYSVALUECHANGED",       /* 0x002d */
"WM_APPTERMINATENOTIFY",    /* 0x002e */
"WM_PRESPARAMCHANGED",      /* 0x002f */

/* control notification messages */

"WM_CONTROL",               /* 0x0030 */
"WM_VSCROLL",               /* 0x0031 */
"WM_HSCROLL",               /* 0x0032 */
"WM_INITMENU",              /* 0x0033 */
"WM_MENUSELECT",            /* 0x0034 */
"WM_MENUEND",               /* 0x0035 */
"WM_DRAWITEM",              /* 0x0036 */
"WM_MEASUREITEM",           /* 0x0037 */
"WM_CONTROLPOINTER",        /* 0x0038 */
"WM_CONTROLHEAP",           /* 0x0039 */
"WM_QUERYDLGCODE",          /* 0x003a */
"WM_INITDLG",               /* 0x003b */
"WM_SUBSTITUTESTRING",      /* 0x003c */
```

```
    "WM_MATCHMNEMONIC",         /* 0x003d */
    "WM_SAVEAPPLICATION",       /* 0x003e */
    "???",

    /* frame manager messages */
    "WM_FLASHWINDOW",           /* 0x0040 */
    "WM_FORMATFRAME",           /* 0x0041 */
    "WM_UPDATEFRAME",           /* 0x0042 */
    "WM_FOCUSCHANGE",           /* 0x0043 */
    "WM_SETBORDERSIZE",         /* 0x0044 */
    "WM_TRACKFRAME",            /* 0x0045 */
    "WM_MINMAXFRAME",           /* 0x0046 */
    "WM_SETICON",               /* 0x0047 */
    "WM_QUERYICON",             /* 0x0048 */
    "WM_SETACCELTABLE",         /* 0x0049 */
    "WM_QUERYACCELTABLE",       /* 0x004a */
    "WM_TRANSLATEACCEL",        /* 0x004b */
    "WM_QUERYTRACKINFO",        /* 0x004c */
    "WM_QUERYBORDERSIZE",       /* 0x004d */
    "WM_NEXTMENU",              /* 0x004e */
    "WM_ERASEBACKGROUND",       /* 0x004f */
    "WM_QUERYFRAMEINFO",        /* 0x0050 */
    "WM_QUERYFOCUSCHAIN",       /* 0x0051 */
    "???",
    "WM_CALCFRAMERECT",         /* 0x0053 */
    "???",
    "WM_WINDOWPOSCHANGED",      /* 0x0055 */
    "???", "???", "???",
    "WM_QUERYFRAMECTLCOUNT",    /* 0x0059 */
    "???",
    "WM_QUERYHELPINFO",         /* 0x005b */
    "WM_SETHELPINFO",           /* 0x005c */
    "WM_ERROR",                 /* 0x005d */
    "???", "???",

/* clipboard messages */
    "WM_RENDERFMT",             /* 0x0060 */
    "WM_RENDERALLFMTS",         /* 0x0061 */
    "WM_DESTROYCLIPBOARD",      /* 0x0062 */
    "WM_PAINTCLIPBOARD",        /* 0x0063 */
    "WM_SIZECLIPBOARD",         /* 0x0064 */
    "WM_HSCROLLCLIPBOARD",      /* 0x0065 */
    "WM_VSCROLLCLIPBOARD",      /* 0x0066 */
    "WM_DRAWCLIPBOARD",         /* 0x0067 */

    "???", "???", "???", "???", "???", "???", "???", "???",

/* keyboard and mouse messages */

    "WM_MOUSEMOVE",             /* 0x0070 */
    "WM_BUTTON1DOWN",           /* 0x0071 */
    "WM_BUTTON1UP",             /* 0x0072 */
    "WM_BUTTON1DBLCLK",         /* 0x0073 */
    "WM_BUTTON2DOWN",           /* 0x0074 */
    "WM_BUTTON2UP",             /* 0x0075 */
    "WM_BUTTON2DBLCLK",         /* 0x0076 */
    "WM_BUTTON3DOWN",           /* 0x0077 */
    "WM_BUTTON3UP",             /* 0x0078 */
    "WM_BUTTON3DBLCLK",         /* 0x0079 */

/* character input messages */
```

```
   "WM_CHAR",                    /* 0x007a */
   "WM_QUEUESYNC",               /* 0x007b */

   "WM_JOURNALNOTIFY",           /* 0x007c */

#define LASTMSG 0x007c
        };
```

The client window procedure has no *switch* structure, since it makes no attempt to take action on the messages it receives. It simply prints out the name of the message, and the message number, in hex. It also prints out "posted" or "sent" to indicate whether the message was posted to the message queue or sent directly to the client window procedure. It knows this because a switch, *route,* is set in the message loop. Following WinGetMsg, *route* is set to "posted", which will be printed out if the window procedure is called by WinDispatchMsg. After WinDispatchMsg returns, *route* is reset to "sent", so this word will be printed out if the window procedure is called directly.

The program beeps when each message arrives, and delays for a half-second using the DosSleep kernel function, so you have time to (very quickly) read the message. When you first start the program, you'll hear the beeping, but you won't see anything because the program has not yet drawn its window and created its presentation space. Soon the window will appear and you'll see the messages fly by. After a while the messages stop. Now move the mouse. You'll see that messages are generated whenever the pointer moves inside the client window. You get different sequences of messages when you position the mouse over the sizing borders than you do for the system menu or title bar. Watch what happens when you select an item from the system menu. Try making the window larger, then smaller. Try maximizing it and minimizing it, and find out what happens when you make it active and inactive.

At this point many of the messages won't mean anything to you. However, as we progress in our study of PM programming, we'll explore additional messages with every new topic. When you're curious about the operation of a message, you can run MESSAGES to see under what circumstances it's generated.

Some message numbers are not defined in the program, even though they are generated by the system and received by our program. (Not all system messages are documented.) We print question marks instead of the message name for these messages.

Since this program draws to the screen on every message, not just on WM_PAINT messages, we can't use the WinBeginPaint and WinEnd-

Paint functions. Instead we use another pair of APIs: WinGetPS and WinReleasePS.

The WinGetPS Function

We use a function called WinGetPS to obtain a cached micro presentation space.

Obtain Cached Micro Presentation Space

```
HPS WinGetPS(hwnd)
HWND hwnd    Window for which presentation space is requested

Returns: Presentation space handle
```

This function returns a handle to the presentation space. This handle will be used by other functions that draw on the PS.

The WinReleasePS Function

When all drawing operations have been completed, and the presentation space is no longer needed by our program, WinReleasePS is used to return the presentation space to PM.

Release a Cached Micro Presentation Space

```
BOOL WinReleasePS(hps)
HPS hps   Presentation space handle

Returns: TRUE if successful, FALSE if error
```

Bad Programming Practice

We should note, somewhat sheepishly, that the MESSAGES program violates one of the cardinal rules of PM programming: don't delay while processing messages. (See the discussion of this under the heading "Messages and Multitasking" earlier in this chapter.) Using the DosSleep kernel function to slow down the message flow is definitely not a good idea if you

plan to run any other programs at the same time. By slowing down the message flow, MESSAGES slows not only itself, but also all the other PM applications running in the system.

Another problem with MESSAGES is that it attempts to write to the screen as soon as the program starts, before a PS has been created. This is why you can hear the beeping before you see anything on the screen. Normally an application should wait until it has a PS before attempting to display anything.

MESSAGES: THE BOTTOM LINE

This chapter has covered a great deal of ground. You've seen how client window procedures handle messages, and you've learned about various theoretical aspects of messages and message flow. What are the key points to know about messages?

On a practical level you need to know how to set up the message queue, message loop, and client window procedure. You need to be aware that your client window procedure will receive messages as the result of various events in the system. Receiving a message is equivalent to being called by another procedure with arguments that specify the message.

Usually a *switch* statement in the client window procedure is used to distinguish one message from another. Depending on the message, further content may need to be extracted from *mp1* and *mp2*, using macros provided in PMWIN.H. Any message not completely handled by the client window procedure should be passed on to WinDefWindowProc for default processing.

On a more theoretical level you should be aware that some messages are sent directly to your client window procedure, while others are posted to your message queue. Messages are read from the queue with WinGetMsg and passed on to the client window procedure with WinDispatchMsg. These two functions are normally placed in a *while* loop.

ON TO THE DETAILS

At this point you've learned most of the fundamentals of PM programming. You know what windows are and what messages are, and you know that windows communicate by transmitting messages to each other. This is probably the toughest aspect of PM programming to assimilate. You're over the hump: From now on everything will be comparatively easy.

6

CONTROLS

We mentioned earlier that PM applications use a different approach to interacting with the user than do traditional command-prompt programs. In a command-prompt environment, such as MS-DOS, the OS/2 kernel, or UNIX, the user and the application typically communicate using short lines of text. How do the user and the application interact in PM? The most common way is with *control windows* (sometimes called simply *controls*). The controls we'll be discussing are predefined public window classes that provide for various kinds of simple user interaction. In contrast to command-prompt programs, controls permit a PM application to conveniently display all the options available when the user must make a choice. Instead of memorizing obscure key-sequences, or looking things up in a manual, the user sees the choices on the screen.

TYPES OF CONTROLS

Different controls are used in different situations. In this section we'll discuss controls from a user's perspective. Then we'll go on to show program examples demonstrating the different types.

Static Controls

A *static* control is a control that does not accept user input. A static control simply writes text (or a graphics element, like an icon) to the screen in a particular place. It's usually used in conjunction with other controls.

Push Buttons

A *push button* is a small rectangle with a three-dimensional appearance, containing text. For example, selecting an option from a program's menu may bring up a window with three push buttons labeled "OK," "Cancel," and "Help."

Clicking on a push button tells the program to take immediate action. It's like a doorbell: you push the button, and something happens.

Radio Buttons

Radio buttons, like the buttons on a car radio, allow you to select one item from a set of mutually exclusive choices. They are used in groups, with only one button in the group being selected at a time. A radio button is an open circle, with text to its right. When you click on a radio button, a black circle appears within it to indicate it is selected. Simultaneously, the previously selected radio button is deselected.

Radio buttons are used to select one of a fairly small number of options. You might use radio buttons to select the parity in a communications program (odd, even, or none), the color of your window, the type of text file to read (ASCII, Word, or WordPerfect), or whether you want your text centered, left justified, or right justified.

Check Boxes

Check boxes allow the user to select one of two states. This could be on or off, color or monochrome, formatted or unformatted, and so on. A check box is a small square with text to its right. When the user clicks on a check box, an "X" appears in it to indicate that it is selected. Check boxes are not logically related, as radio buttons are. A single check box can appear by itself, and even when check boxes are grouped visually, each one is checked or unchecked independently of the others.

In a spreadsheet program, a check box could be used to select immediate recalculation (as opposed to recalculation only on command). A word processing program might use check boxes to select characteristics of the text. Three check boxes would select bold or normal, underlined or normal, and italic or normal. The use of check boxes allows a combination of these

characteristics: text can be bold *and* underlined, while radio buttons permit only one selection.

There is also a three-state check box that can be blank, checked, and also shaded, which indicates an indeterminate state. These are less commonly used.

Entry Fields

An *entry field* is a box into which the user can type a one-line string of characters. Entry fields are typically used to input numbers or short text strings, such as a margin setting in inches, a page number, a word to be searched for, or a pathname. An entry field is used where it is impractical to show all the possible choices.

Entry fields should be used only when necessary. They don't permit the user to choose from a list of options displayed on the screen; the user must know what to enter. If you use an entry field, make sure it's obvious what the user should type, and try to minimize the amount of typing necessary.

List Boxes

A *list box* contains a list of items, each represented by a word or phrase. The user selects the item by clicking on this text. If there are too many items to fit in the box, a scroll bar to the right of the list permits the user to scroll the list up and down in the list box window. A list box might be used to hold the data files accessible to a database program, a list of fonts, a list of printers, and so on.

You can program the list box to allow the user to select more than one item at a time by dragging the pointer across several items. If file names were being displayed, for example, this feature might allow you to delete or copy several files at once.

Since it can be scrolled, a list box can hold many more items than would fit conveniently on the screen. Also, the size of the list box, and hence the screen appearance, doesn't change if the application adds or deletes items from the list. This gives list boxes more flexibility than other kinds of controls such as radio buttons.

Scroll Bars

Scroll bars allow a selection to be made from a continuous range of values. They are typically used to scroll text in a window, but they can also be used to position graphics items, select a color from a spectrum of colors, and set such continuously varying numbers as the cursor blink rate.

Scroll bars can be stand-alone controls like those discussed previously, but they are more commonly parts of a standard window, as the title bar and and System Menu are. The various kinds of controls are shown in Figure 6-1.

Other Controls

There are two other controls that are somewhat more complex: *combo boxes* and *multi-line edit* controls (MLEs). A combo box is an edit control with a pull-down list of choices. An MLE is a multi-line entry field box. We won't discuss these controls.

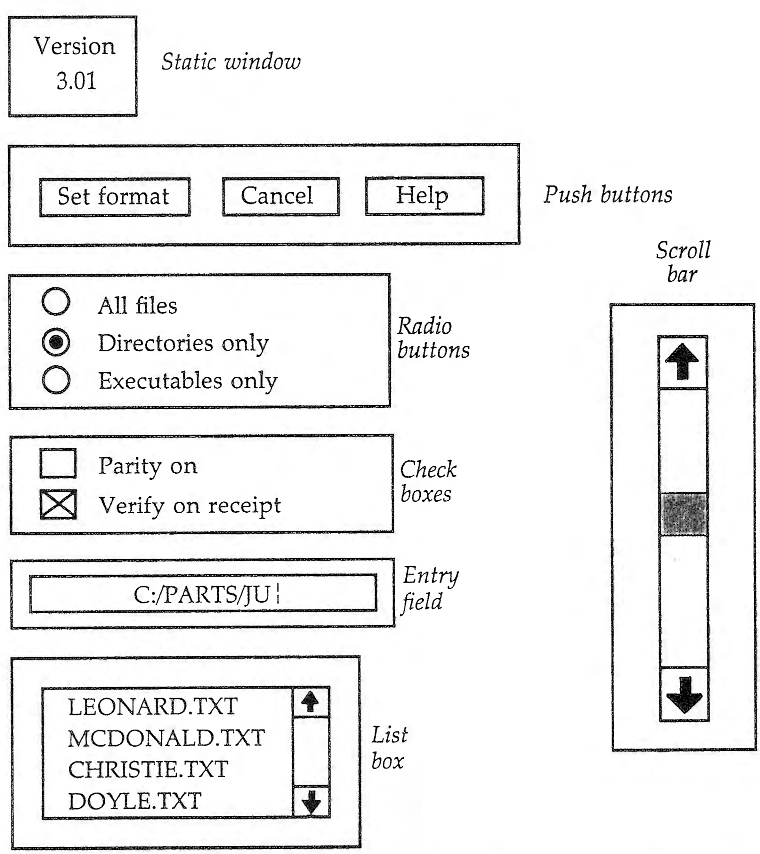


Figure 6-1. Control window types

Menus are a kind of control. However, menus are almost always used in conjunction with resources and require considerable discussion. They're covered in Chapter 8.

You can also define your own controls or purchase controls created and marketed by different firms. We won't explore these possibilities here.

Notes on Controls

The control window procedure itself takes care of many of the clerical details of operating the control. For instance, when you click on a push button, the button control procedure appears to depress the button so you know it's responding to you. Your program doesn't need to worry about doing this. Instead, you receive messages only when a noteworthy event occurs: something your program will be interested in, such as the fact that a particular button was pressed. This makes it much simpler to program controls.

Sometimes two or more controls seem equally appropriate in a particular situation. Choosing the right one may involve a subtle perception of how the user interface should be designed, and may come down to a question of individual style.

Controls are often created with WinCreateWindow, as they are in most of the examples in this chapter. They can also be created as elements of dialog boxes. Dialog boxes are usually constructed as elements of resources, so we won't discuss them until Chapter 9.

TEXT IN A STATIC CONTROL

Our first example demonstrates the static control window. This is the simplest control. While most controls accept user input, this one can only write to the screen. Perhaps "control" is an inappropriate term for static controls, since they don't offer the user the chance to control the program. However, they are created in the same way as other control windows.

Our example program uses a static control to write the phrase "This is the text in the static window" to the screen whenever mouse button 1 is clicked in the client window. The window containing this phrase will be displayed wherever the mouse pointer is.

User-Defined Header Files

We've added an extra file to this program: STATIC.H. It's common practice in larger C programs to place certain kinds of data in header files. This data often consists of *#define* and *#include* directives of various kinds. Placing

such directives in a header file permits their access by all files in applications with multiple source files. Header files are often used in PM programs because so many *#define* directives are used for windows and similar entities. (This is especially true of the large number of ID numbers involved in resources.) Header files also serve as a convenient place for function prototypes. You should add the header file, STATIC.H in this example, to the Make file. STATIC.OBJ is dependent on it.

In the example programs in this chapter, header files serve another purpose: they permit us to use the same *main,* Make, and definition files for each example in the chapter. Only the client window procedure and the header file vary from example to example. This approach allows us to save space and to focus attention on the part of the program under discussion.

The STATIC Program

Here are the listings for STATIC.C, STATIC.H, STATIC, and STATIC.DEF.

```
/* --------------------------- */
/* STATIC.C - A static control */
/* --------------------------- */

#define INCL_WIN
#define INCL_GPI
#include <os2.h>

#include "static.h"

USHORT cdecl main(void)
   {
   HAB hab;                           /* handle for anchor block */
   HMQ hmq;                           /* handle for message queue */
   QMSG qmsg;                         /* message queue element */
   HWND hwndFrame, hwndClient;        /* handles for windows */

                                      /* create flags for std window */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);         /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);   /* create message queue */

                                      /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                      /* create standard window */
   hwndFrame = WinCreateStdWindow (HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
               szClientClass, " - Controls", 0L, NULL, 0, &hwndClient);

                                      /* message loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
```

```
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwndFrame);        /* destroy frame */
   WinDestroyMsgQueue(hmq);            /* destroy message queue */
   WinTerminate(hab);                  /* terminate PM usage */

   return 0;
   }

                                       /* client window procedure */
MRESULT EXPENTRY ClientWinProc (HWND hwnd, USHORT msg, MPARAM mp1,
                     MPARAM mp2)
   {
   HWND hwndControl;                   /* control window handle */

   switch (msg)
      {
      case WM_BUTTON1DOWN:
         hwndControl = WinCreateWindow(
                          hwnd,         /* parent is client window */
                          WC_STATIC,    /* a static control window */
                          "This is the text in the static window",
                          WS_VISIBLE |
                          WS_CLIPSIBLINGS |  /* clip to top siblings */
                          SS_TEXT |          /* text */
                          DT_LEFT |          /* left justify */
                          DT_TOP |           /* from top of window */
                          DT_WORDBREAK,      /* word wrap */
                          SHORT1FROMMP(mp1), SHORT2FROMMP(mp1), 60, 80,
                          hwnd,              /* owner */
                          HWND_TOP, ID_WINDOW, NULL, NULL);

         return FALSE;                 /* continue window creation */
         break;

      case WM_ERASEBACKGROUND:
         return TRUE;                 /* erase background */
         break;

      default:                         /* default window processing */
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                        /* NULL / FALSE */
   }
```

```
/* -------- */
/* STATIC.H */
/* -------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_WINDOW 1
```

```
# ----------------
# STATIC make file
# ----------------

static.obj: static.c static.h
   cl -c -G2s -W3 -Zp static.c

static.exe: static.obj static.def
   link /NOD static,,NUL,os2 slibce,static
```

```
; ----------
; STATIC.DEF
; ----------

NAME         STATIC   WINDOWAPI

DESCRIPTION 'A static control'

PROTMODE
STACKSIZE   4096
```

When you run this program, the usual top-level window will appear. If you click anywhere in the client window, a window with the phrase "This is the text in the static window" will appear at that point. You can create as many copies of this text as you like by clicking repeatedly in the client window.

The Make and definition files for STATIC are similar to those in previous examples. The *main* function in STATIC.C is almost the same as in previous examples, but includes two additional lines:

```
#define INCL_GPI

#include "static.h"
```

The first line causes the file PMGPI.H to be included. This file is not needed in this program, but will be used in other examples in this chapter. Including it here permits us to use the same *main* function for all programs.

The second line causes the user-defined header file, discussed above, to be included.

The Standard Window

The *main* routine in STATIC creates a standard window. Almost all PM programs start with at least one top-level standard window, so that's what we'll do in most of our examples. The standard window has a title bar,

system menu, sizing border, and minimize and maximize icons, as specified by the *pflCreateFlags* argument to WinCreateStdWindow. It also has a client window. Most of the control windows we will create in this chapter will be children of the client window.

Creating Static Controls with WinCreateWindow

The client window procedure for this example is fairly simple. It creates a static control window, using WinCreateWindow, whenever the WM_BUTTON1DOWN message is received. The WinCreateWindow function was discussed in Chapter 4. However, several of its arguments are given different values in this example. Let's see what these values are, and how they're used to create a control window.

Class

The class of window is determined by the WC_ identifier in the *pszClass* argument. For a static window we use WC_STATIC. Other examples of WC_ identifiers for control windows are shown in the following table:

Identifier	Control Window
WC_STATIC	Static window
WC_BUTTON	Buttons (push button, radio button, check box)
WC_LISTBOX	List box
WC_SCROLLBAR	Scroll bar
WC_ENTRYFIELD	Entry field
WC_TITLEBAR	Title bar
WC_MENU	Menu
WC_MLE	Multi-line edit field
WC_COMBOBOX	Combo Box

We'll see examples of other control window classes later in this chapter.

Name

The third argument to WinCreateWindow, *pszName,* contains the text to be placed in the static window. In our example it's "This is the text in the static window".

Style

The *flStyle* argument has many components, ORed together. These identifiers are defined in PMWIN.H. We discussed WS_VISIBLE in Chapter 4, and we'll look at WS_CLIPSIBLINGS in the next section.

The SS_TEXT identifier specifies text. Other possibilities for static controls include SS_ICON, SS_BITMAP, and SS_GROUPBOX.

DT_ identifiers specify aspects of the text and how it's displayed in the static window. They're used with SS_TEXT.

Identifier	Meaning
DT_LEFT	Left justify
DT_RIGHT	Right justify
DT_CENTER	Center horizontally
DT_TOP	Start at top
DT_BOTTOM	Start at bottom
DT_VCENTER	Center vertically
DT_HALFTONE	Use halftone text
DT_WORDBREAK	Break text at word boundaries
DT_ERASERECT	Erase rectangle before writing text

We use DT_LEFT, DT_TOP, and DT_WORDBREAK to left justify the text, start it at the top of the window, and cause it to break at word boundaries. (DT_WORDBREAK can only be used when you specify DT_TOP and DT_LEFT.)

Clipping Siblings

To see why you need the WS_CLIPSIBLINGS identifier, perform the following experiment. Click repeatedly in the client window, so that the copies of the phrase overlap each other. The phrases generated by later clicks will overlay those generated by earlier clicks. We say that the phrases generated later have a higher *Z-axis* ordering; that is, they appear to be on top of other windows, or closer to the person viewing the screen.

Now minimize the window, and expand it again. You'll see that, surprisingly, the windows are drawn in the reverse order to the way they were created. However, they will overlap in the same way so the picture looks as it did originally, as shown in Figure 6-2.

Now take the WS_CLIPSIBLINGS identifier out of the program and recompile. Minimize and expand the window. This time, the windows drawn earlier will cover those drawn later. The result won't look like the original. What's happening?

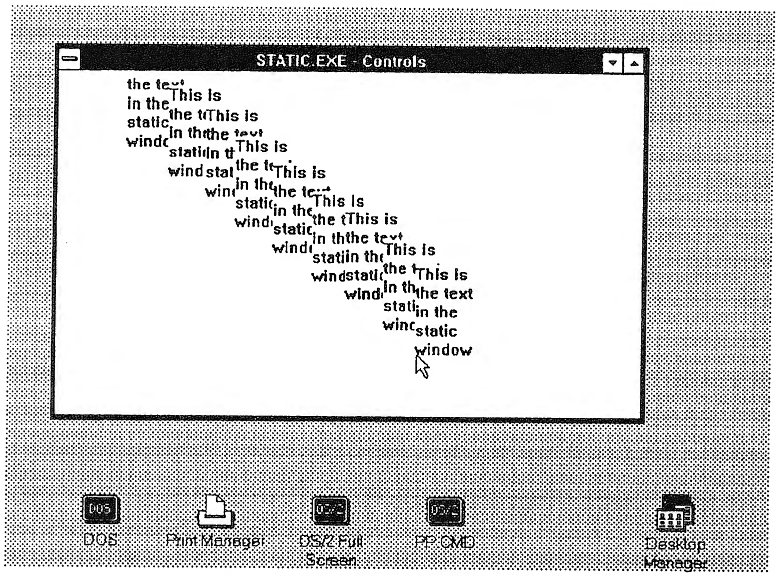


Figure 6-2. Output of STATIC program

Children are always clipped to their parent, but in the absence of specific instructions, PM doesn't clip siblings to each other. This optimizes performance, and if you plan your display so that siblings don't overlap, it causes no harm. If siblings will overlap, then you need to give more specific instructions to PM.

The WS_CLIPSIBLINGS identifier in WinCreateWindow will ensure that whenever a window is drawn it will *clip itself to siblings with higher Z-axis ordering.* Then even if it is redrawn later, the windows will overlap in the correct way.

Position and Size

The coordinates for positioning the static window are acquired from *mp1*, which, for a WM_BUTTON1DOWN message, conveys the mouse coordinates at the time button 1 was pressed.

The static window is made 60 pixels wide and 80 pixels high. How do we know how large to make the window? We want it to be just big enough to hold the text to be displayed. We've cheated a little by using trial and error to find appropriate dimensions. In a full-scale application, the program would query the system to find the size of the font and the length of

the text. This would ensure that the box was exactly the right size to hold the text, even if the system font were changed. We'll touch on this point again in the SCROLL example at the end of this chapter and explore it more thoroughly in Chapter 11, when we discuss fonts.

Other WinCreateWindow Arguments

The owner of the static window (the ninth argument to WinCreateWindow) is the same as its parent: the client window. Ownership is important with control windows, since it is *to its owner* that the control sends messages when something noteworthy happens.

The value for *hwndInsertBehind* is HWND_TOP, which places each control on top of its preexisting siblings. The 11th argument, *id,* is given the value WINDOW_ID, which we *#define*d in the header file STATIC.H. We don't use this argument in the present example, but we'll see later how it's used to distinguish different windows. The last two arguments to WinCreateWindow, *pCtlData* and *pPresParams,* are not used.

Multiple Instances

You may have noticed something unusual about this program: you can make as many static windows as you want, but there is only one WinCreateWindow call in the listing. Each time you click mouse button 1, the client window procedure is called, and another instance of the static window is created. All these static windows, even if you put dozens of them on the screen at once, are defined by the same client window procedure. They are all instances of the static control window class, and they all have the same owner: the client window procedure.

Messages in STATIC

The only messages explicitly handled by the client window procedure in STATIC are WM_BUTTON1DOWN and WM_ERASEBACKGROUND. We've seen in previous examples how WM_ERASEBACKGROUND is used, and you know that WM_BUTTON1DOWN is generated whenever the left mouse button is pressed. In our program a new instance of the static window is created each time this message is received.

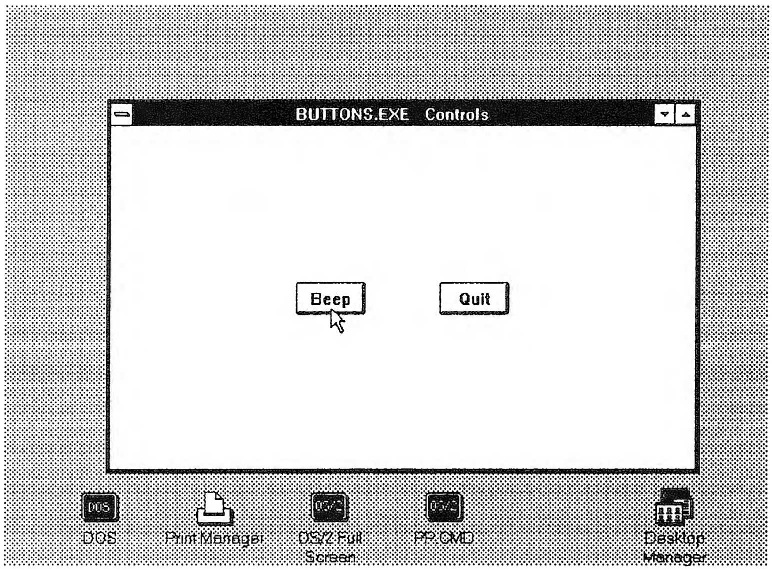


Figure 6-3. Output of BUTTONS program

Static windows don't send messages to their owner, the client window procedure. Other control windows send and receive messages from their owners, as we'll see in the following examples.

TAKING ACTION WITH PUSH BUTTONS

A push button consists of a rectangular outline drawn around a word or a short phrase. This phrase describes an action. Clicking on the button causes the action specified to be carried out.

Our example places two buttons, labeled "Beep" and "Quit," in the middle of the screen, as shown in Figure 6-3.

As you might guess, clicking on one of these buttons sounds a tone, while clicking on the other causes the program to terminate.

Here are the listings for the client window procedure from BUTTONS.C, and for BUTTONS.H.

```
                                    /* client window procedure */
MRESULT EXPENTRY ClientWinProc (HWND hwnd, USHORT msg, MPARAM mp1,
                   MPARAM mp2)
   {
   HWND hwndControl1, hwndControl2;    /* control windows handles */
   RECTL rcl;
```

```
SHORT cx, cy, x, y;

switch (msg)
   {
   case WM_CREATE:
                                     /* post private create msg */
      WinPostMsg(hwnd, CWPM_CREATE, NULL, NULL);
      return FALSE;                  /* continue window creation */
      break;

   case CWPM_CREATE:                 /* private create msg */
      x = 60; y = 30;                /* set window size */
                                     /* calculate button position */
      WinQueryWindowRect (hwnd, &rcl);
      cx = (SHORT)((rcl.xRight - rcl.xLeft) / 2 - x - x / 2);
      cy = (SHORT)((rcl.yTop - rcl.yBottom) / 2 - y / 2);

      hwndControl1 = WinCreateWindow(    /* create a new window */
                        hwnd,
                        WC_BUTTON,       /* button control */
                        "Beep",
                        WS_VISIBLE |
                        BS_PUSHBUTTON,   /* pushbutton style */
                        cx, cy, x, y,
                        hwnd, HWND_TOP, ID_BUTTON1, NULL, NULL);

      cx += x; cx +=  x;             /* calculate button position */

      hwndControl2 = WinCreateWindow(    /* create a new window */
                        hwnd,
                        WC_BUTTON,       /* button control */
                        "Quit",
                        WS_VISIBLE |
                        BS_PUSHBUTTON,   /* pushbutton style */
                        cx, cy, x, y,
                        hwnd, HWND_TOP, ID_BUTTON2, NULL, NULL);
      break;

   case WM_COMMAND:                  /* check for a button push */
      if (SHORT1FROMMP(mp2) == CMDSRC_PUSHBUTTON)
         {
         if (SHORT1FROMMP(mp1) == ID_BUTTON1)   /* button 1 */
            WinAlarm(HWND_DESKTOP, WA_NOTE);    /* sound Beep */
         else
            if (SHORT1FROMMP(mp1) == ID_BUTTON2)        /* button 2 */
               WinPostMsg(hwnd, WM_QUIT, NULL, NULL);   /* Quit */
         }
      break;

   case WM_ERASEBACKGROUND:
      return TRUE;                   /* erase background */
      break;

   default:                          /* default window processing */
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }

return NULL;                         /* NULL / FALSE */
}
```

```
/* --------- */
/* BUTTONS.H */
/* --------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_BUTTON1 1
#define ID_BUTTON2 2
#define CWPM_CREATE WM_USER
```

The *main* function from BUTTONS.C, the Make file BUTTONS, and the definition file BUTTONS.DEF are similar to STATIC and STATIC.DEF; only the program name needs to be changed wherever it appears in these files. This includes the one line in *main* that should be modified:

```
#include "static.h"
```

should be changed to:

```
#include "buttons.h"
```

Creating Buttons with WinCreateWindow

There are two calls to WinCreateWindow in the client window procedure: one for each button. Let's look at the arguments needed to create push buttons.

The class identifier is WC_BUTTON, which is used for buttons of various types. The *idName* argument is the text that will appear in the button. The style includes the identifier BS_PUSHBUTTON. This defines the type of button. Other possibilities are shown in the following table:

Identifier	**Purpose**
BS_PUSHBUTTON	Push button
BS_CHECKBOX	Check box
BS_AUTOCHECKBOX	Automatic check box
BS_RADIOBUTTON	Radio button
BS_AUTORADIOBUTTON	Automatic radio button
BS_3STATE	Three-state button
BS_AUTO3STATE	Automatic three-state button
BS_USERBUTTON	User-defined button

How big should we make the buttons? As in the last example, we use trial and error to establish that 60 pixels wide and 30 high is large enough to hold the text in both buttons. To position the buttons, we call WinQueryWindowRect to find the size of the client window, and use the resulting dimensions to calculate the center of the screen. From the center, appropriate offsets are added for each button, as shown in Figure 6-4.

The *id* arguments to the two WinCreateStdWindow calls are set to ID_BUTTON1 and ID_BUTTON2, respectively. These identifiers, *#define*d in BUTTONS.H, will be used later in the program.

Messages in BUTTONS

Let's look at the messages handled by BUTTONS and see what action is taken for each one.

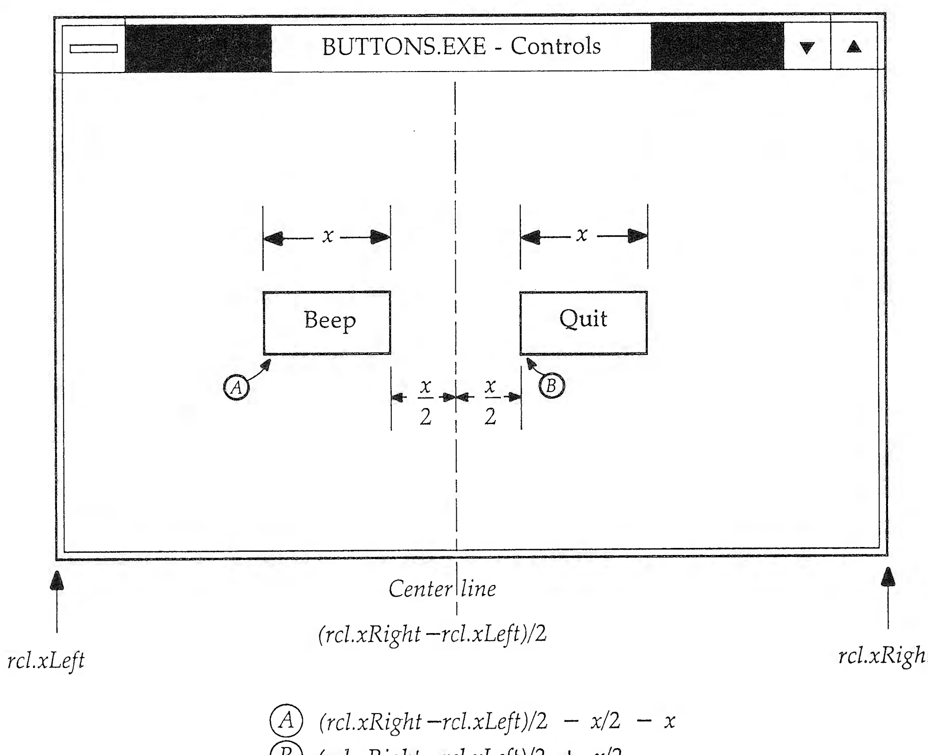


Figure 6-4. Positioning of push buttons in BUTTONS

WM_CREATE and CWPM_CREATE

It makes sense for the client window procedure to create the control windows that it will own (push buttons in this example). Here the client window procedure creates its controls as soon as it is created itself.

A message, WM_CREATE, is sent to a window to announce that it has been created. If this message worked as you might expect, you could simply wait for it to arrive, and create the control windows at that time. However, there is a problem. While the client window may have been created, it still has zero size when WM_CREATE is received. When the controls are created, they should be positioned (and sometimes sized) based on the dimensions of the client window. But attempting to find the dimensions of the client window with WinQueryWindowRect when receiving the WM_CREATE message results in zero values.

Our solution to this problem is to have the client window procedure post another message to itself when it receives WM_CREATE. Receipt of this second message is then used as the signal to create the control windows. The second message can arrive only after the first message has been processed; that is, after the client window has been created and sized.

For this second message a *user-defined* message is used. User-defined messages have numbers starting from WM_USER, which is specified in PMWIN.H. We use WM_USER and redefine it in the BUTTONS.H header file as CWPM_CREATE (for client window procedure message), which is more self-explanatory.

As soon as this message is received, the two push buttons are created, using WinCreateWindow.

Owners and Ownees

We've mentioned that when a noteworthy event occurs to it, a control window transmits messages to its owner, not to its parent. In the examples in this chapter the client window procedure is both the owner and the parent of the control windows we discuss. However, in some situations the owner and parent are different windows: The control is displayed within the parent, but sends its messages to its owner. The only requirement for ownership is that the owner and ownee must be in the same thread.

The major difference between parenthood and ownership is that parenthood defines clipping and painting, while ownership defines communication between windows.

The WM_COMMAND Message

A control window often uses the WM_COMMAND message to inform its owner that a selection has been made. In BUTTONS there are two control windows, both push buttons. They send the WM_COMMAND message to their owner, the client window, when they are clicked with the mouse.

The *mp1* and *mp2* message parameters reveal additional information about WM_COMMAND, as shown in this table:

Identifier	Meaning
SHORT1FROMMP(mp1)	Control window ID
SHORT2FROMMP(mp1)	0
SHORT1FROMMP(mp2)	CMDSRC_ identifier
SHORT2FROMMP(mp2)	0=keyboard input, non-zero=mouse

The first half of *mp2* identifies the source of the WM_COMMAND message. There are four possibilities:

Identifier	Source of Command
CMDSRC_PUSHBUTTON	Push button
CMDSRC_MENU	Menu
CMDSCR_ACCELERATOR	Keyboard accelerator
CMDSRC_OTHER	Other

In our example we're looking for commands from push buttons, so we check for CMDSRC_PUSHBUTTON in the first *if* statement.

The first half of *mp1* is the ID of the control window sending the message. Our example uses this value to determine which of the two buttons has been pressed. The macro SHORT1FROMMP, described in the last chapter, extracts the ID from *mp1,* and the *if else* statements take different action depending on the push button ID.

SELECTING WITH RADIO BUTTONS

Radio buttons permit the user to select one of a number of choices. Our next example program allows the user to change the color of the desktop window. It displays three radio buttons, one with the text "Red," one with "Green," and one with "Blue," as shown in Figure 6-5.

Clicking on a radio button immediately changes the desktop to the color selected. Radio buttons sometimes produce immediate effects like this, but may also make a choice that manifests itself only later.

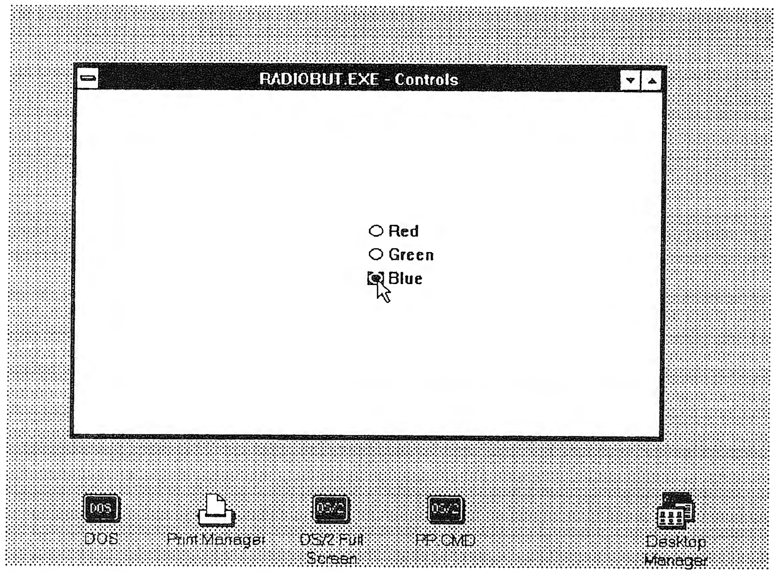


Figure 6-5. Output of RADIOBUT program

When you exit this program, the original desktop color is restored. Here are the listings for the client window procedure from RADIOBUT.C, and for RADIOBUT.H. The *main* function and the Make and definition files are similar to those for the STATIC example.

```
                                         /* client window procedure */
MRESULT EXPENTRY ClientWinProc (HWND hwnd, USHORT msg, MPARAM mp1,
                     MPARAM mp2)
   {
                                         /* control windows handles */
   HWND hwndControl1, hwndControl2, hwndControl3;
   RECTL rcl;
   SHORT cx, cy;
   COLOR aclrIndRGB[2];
   static COLOR clrOldIndRGB;

   aclrIndRGB[0] = SYSCLR_BACKGROUND;  /* ref. system background color */

   switch (msg)
      {
      case WM_CREATE:
                                         /* post private create msg */
         WinPostMsg(hwnd, CWPM_CREATE, NULL, NULL);
         return FALSE;                   /* continue window creation */
         break;

      case CWPM_CREATE:                  /* private create msg */
                                         /* calculate center of window */
         WinQueryWindowRect (hwnd, &rcl);
         cx = (SHORT)((rcl.xRight - rcl.xLeft) / 2);
```

```
      cy = (SHORT)((rcl.yTop - rcl.yBottom) / 2);

                                    /* create red button */
      hwndControl1 = WinCreateWindow(
                        hwnd,
                        WC_BUTTON,  /* a button window */
                        "Red",
                        WS_VISIBLE ¦
                        BS_AUTORADIOBUTTON,   /* auto radio button */
                        cx, cy + 20, 60, 15,
                        hwnd, HWND_TOP, ID_BUTTON1, NULL, NULL);

                                    /* create green button */
      hwndControl2 = WinCreateWindow(hwnd, WC_BUTTON, "Green",
                        WS_VISIBLE ¦ BS_AUTORADIOBUTTON,
                        cx, cy, 60, 15,
                        hwnd, HWND_TOP, ID_BUTTON2, NULL, NULL);

                                    /* create blue button */
      hwndControl3 = WinCreateWindow(hwnd, WC_BUTTON, "Blue",
                        WS_VISIBLE ¦ BS_AUTORADIOBUTTON,
                        cx, cy - 20, 60, 15,
                        hwnd, HWND_TOP, ID_BUTTON3, NULL, NULL);

                              /* save current system background color */
      clrOldIndRGB = WinQuerySysColor(HWND_DESKTOP, SYSCLR_BACKGROUND,
                        0L);
      break;

   case WM_CONTROL:                 /* button pushed */
      switch SHORT1FROMMP(mp1)
         {
         case ID_BUTTON1: aclrIndRGB[1] = RGB_RED;
         break;
         case ID_BUTTON2: aclrIndRGB[1] = RGB_GREEN;
         break;
         case ID_BUTTON3: aclrIndRGB[1] = RGB_BLUE;
         break;
         }
                                    /* set background color */
      WinSetSysColors(HWND_DESKTOP, 0L, LCOLF_INDRGB, 0L, 2L,
         aclrIndRGB);
      break;

   case WM_DESTROY:                 /* window termination */
      aclrIndRGB[1] = clrOldIndRGB;
      WinSetSysColors(HWND_DESKTOP, 0L, LCOLF_INDRGB, 0L, 2L,
         aclrIndRGB);
      break;

   case WM_ERASEBACKGROUND:
      return TRUE;                  /* erase frame's client area */
      break;

   default:                         /* default window processing */
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }

return NULL;                        /* NULL / FALSE */
}
```

```
/* ---------- */
/* RADIOBUT.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_BUTTON1 1
#define ID_BUTTON2 2
#define ID_BUTTON3 3

#define CWPM_CREATE WM_USER
```

Creating Radio Buttons with WinCreateWindow

As we did for push buttons, we use the WC_BUTTON identifier to specify the window class. All buttons belong to this class. The *type* of button is BS_AUTORADIOBUTTON. With automatic radio buttons the button window procedure takes care of deactivating the previously active button when a new button is clicked on. Normal radio buttons, specified by BS_RADIOBUTTON, require the program to deactivate the previously active button.

The WM_CONTROL Message

A control window uses a WM_CONTROL message to inform its owner that something noteworthy has happened. Unlike the message WM_COMMAND, which is *posted*, WM_CONTROL is always *sent*.

When WM_CONTROL is received, the *mp1* parameter specifies the control window ID. In RADIOBUT we use the identifiers ID_BUTTON1, ID_BUTTON2, and ID_BUTTON3 for the three radio button controls. We then use a *switch* statement to take different actions depending on which of these ID values is returned in *mp1*.

The WM_DESTROY Message

The WM_DESTROY message is transmitted by PM when a window is about to be destroyed. It can be used by a window procedure to perform cleanup and termination chores. In RADIOBUT we use it to restore the screen background color to its original value before the program terminates.

Querying and Setting System Colors

System colors are the colors used for the system-created visual features of the PM environment, such as windows, menus, scroll bars, sizing borders, and various kinds of text. These colors can be changed, as you know if

you've played with the Control Panel utility. WinQuerySysColor is used to find the color of a particular feature, and WinSetSysColors is used to change the color of one or more features.

In RADIOBUT we obtain the color of the screen background when we receive the WM_CREATE message. We store this color and set it again on exiting the program.

Return Specified System Color

```
COLOR WinQuerySysColor(hwndDesktop, clr, lReserved)
HWND hwndDesktop          Desktop window handle (HWND_DESKTOP)
COLOR clr                 System color index value (SYSCLR_)
LONG lReserved            Reserved; must be zero

Returns: RGB color value corresponding to index value in clr
```

WinQuerySysColor is given an identifier that specifies the particular feature whose color we want to know. Examples are SYSCLR-_SCROLLBAR, SYSCLR_BACKGROUND, and SYSCLR_MENU. These are defined in PMWIN.H. WinQuerySysColor then returns the color value of the feature. In RADIOBUT we save the original color value for SYSCLR_BACKGROUND in the variable *clrOldIndRGB.*

WinSetSysColors is a versatile function. It can set all the system colors at once, or only some of them, and it can use color tables in several formats. In RADIOBUT we use it to set only one color.

Set System Colors

```
BOOL WinSetSysColors(hwndDesktop, flOptions, flFormat, clrFirst,
                     cclr, pclr)
HWND hwndDesktop          Desktop window handle (HWND_DESKTOP)
ULONG flOptions           Specify default reset and dithering
ULONG flFormat            Format of color table
COLOR clrFirst            Starting system color index
ULONG cclr                Number of elements in color table
PLONG pclr                Address of color table

Returns: TRUE if successful, FALSE if error
```

We specify the color table format using the LCOLF_INDRGB identifier in the third argument. This format consists of pairs of COLOR values. The first value is the index of the feature whose color will be set, and the second is the color value. We're only setting one color, so we use an array *clrIndRGB* of two elements to hold the index and color values. We set the index when the client window procedure is first called, with the line

```
clrIndRGB[0] = SYSCLR_BACKGROUND;
```

The color values are set in the *switch* statement, depending on which of the three radio buttons is pressed, in lines like

```
case BUTTON1_ID: clrIndRGB[1] = RGB_RED;
```

WinSetSysColors is then used to change the background color, which is immediately evident on the screen. When the WM_DESTROY message is received, WinSetSysColors is invoked again to restore the background to its original color.

Static Variables

Notice that the variable *clrOldIndRGB,* which retains the original background color value, is of type *static.* This variable must be made *static* so that it will preserve its value between messages; that is, between calls to ClientWinProc.

TOGGLING WITH CHECK BOXES

A *check box* is essentially a switch that can be turned on and off. It appears as a small square with text to its right. The *on* condition is indicated by a cross in the check box, the *off* condition by the absence of a check.

Our example program creates a single check box. This check box is used to specify whether the program will beep when it receives a WM_PAINT message. The screen generated by CHECKBOX is shown in Figure 6-6.

Here are the listings for the client window procedure from CHECKBOX.C, and CHECKBOX.H. The *main* function and the Make and definition files are similar to those in the STATIC example.

```
                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc (HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   static HWND hwndControl;             /* control window handle */
   RECTL rcl;
   static SHORT fsButtonState = FALSE;  /* button not checked */
   SHORT cx, cy;

   switch (msg)
      {
      case WM_CREATE:
                                        /* post private create msg */
         WinPostMsg(hwnd, CWPM_CREATE, NULL, NULL);
         return FALSE;                  /* continue window creation */
         break;
```

```
    case CWPM_CREATE:                 /* private create msg */

                                      /* calculate center of window */
       WinQueryWindowRect (hwnd, &rcl);
       cx = (SHORT)((rcl.xRight - rcl.xLeft) / 2);
       cy = (SHORT)((rcl.yTop - rcl.yBottom) / 2);

       hwndControl = WinCreateWindow(
                          hwnd,
                          WC_BUTTON,  /* a button window */
                          "Beep on WM_PAINT",
                          WS_VISIBLE |
                          BS_AUTOCHECKBOX,  /* an automatic check box */
                          cx - 100, cy - 10, 200, 20,
                          hwnd, HWND_TOP, ID_BUTTON, NULL, NULL);
       break;

    case WM_CONTROL:                  /* button state changed */
       fsButtonState = SHORT1FROMMR(WinSendMsg(hwndControl,
                                 BM_QUERYCHECK, NULL, NULL));
       break;

    case WM_PAINT:
                            /* beep on WM_PAINT msg if box is checked */
       if (fsButtonState) WinAlarm(HWND_DESKTOP, WA_NOTE);
       return(WinDefWindowProc(hwnd, msg, mp1, mp2));
       break;

    case WM_ERASEBACKGROUND:
       return TRUE;                   /* erase frame's client area */
       break;

    default:                          /* default window processing */
       return(WinDefWindowProc(hwnd, msg, mp1, mp2));
       break;
    }

 return NULL;                         /* NULL / FALSE */
 }
```

```
/* ---------- */
/* CHECKBOX.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_BUTTON 1
#define CWPM_CREATE WM_USER
```

Run the program, and, without checking the box, use the sizing border to make the window larger and smaller. No beeps occur. Click on the check box, which is labeled "Beep on WM_PAINT." A cross will appear in the box. Now make the window larger and smaller. You will hear the beep

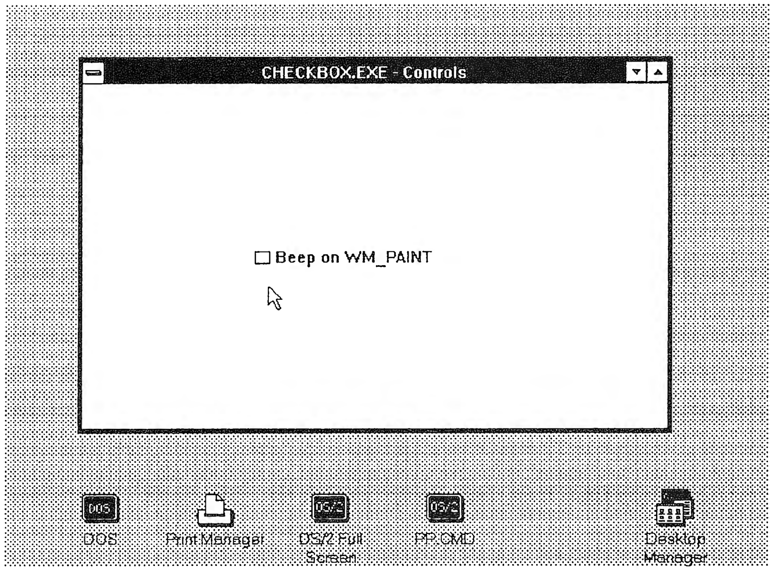


Figure 6-6. Output of CHECKBOX program

whenever you resize the window, and also when the window is maximized and minimized. Experimenting with this program will give you a feeling for the WM_PAINT message, as well as for check boxes.

Creating Check Boxes with WinCreateWindow

A check box is a kind of button, so the arguments to WinCreateWindow are similar to those for push buttons and radio buttons. All buttons use the WC_BUTTON class. The identifier BS_AUTOCHECKBOX in the *flStyle* argument specifies an automatic check box.

The check box is initially positioned in the center of the client window, using coordinates obtained with WinQueryWindowRect. Its size includes its associated text and is made 200 pixels long and 20 pixels high.

Querying the Check Box

A WM_CONTROL message is received whenever the user toggles the state of the check box between checked and unchecked. When the program receives a WM_CONTROL, it queries the state of the button by sending a

BM_QUERYCHECK message back to the check box. This message asks the check box whether it is checked. The return value from WinSendMsg is 0 if the box is unchecked, 1 if it is checked, and 2 if its state is indeterminate. In CHECKBOX this return value is stored in the variable *fsButtonState,* which has the storage class *static,* so it will retain its value between calls to ClientWinProc.

A message is usually posted unless there's a reason to send it. In this case we send the BM_QUERYCHECK message because we need to wait for a reply from the check box before we can continue processing of the WM_CONTROL messages. (See the discussion of sending and posting in the preceding chaper.)

The WM_PAINT message causes WinAlarm to be executed only if the *fsButtonState* variable is set to 1, indicating the box is checked. Whether or not WinAlarm is executed, WinDefWindowProc is used to perform the default activities normally associated with WM_PAINT. This involves executing WinBeginPaint and WinEndPaint, which, as we noted in Chapter 5, is commonly performed when a WM_PAINT message is received.

TYPING INTO ENTRY FIELDS

One of the more traditional ways to input information to a program is to type it in. PM makes this possible with the entry field control. Entry fields are rectangular boxes into which the user can type a single line of text or numbers. The user can backspace to delete text, and reposition the cursor to insert text anywhere in the field. The user can also select (highlight) the text by dragging across it with the mouse and press the DEL or BACKSPACE key to delete it. When the entry field first appears, it may already contain text, which the user can modify.

Our example program creates an entry field. To show how the text is extracted from the field, the program copies the text to the screen above the entry field. Typical output is shown in Figure 6-7.

Here are the ENTRY.C and ENTRY.H files. Note that a complete new .C file is used, not just a different client window procedure. The Make and definition files are similar to those in the STATIC example.

```
/* -------------------------------- */
/* ENTRY.C - An entry field control */
/* -------------------------------- */

#define INCL_WIN
#include <os2.h>
```

```
#include "entry.h"

USHORT cdecl main(void)
   {
   HAB hab;                           /* handle for anchor block */
   HMQ hmq;                           /* handle for message queue */
   QMSG qmsg;                         /* message queue element */
   HWND hwndFrame, hwndClient;        /* handles for windows */

                                      /* create flags for std window */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);         /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);   /* create message queue */

                                      /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW |
                    CS_CLIPCHILDREN,
                    0);

                                      /* create standard window */
   hwndFrame = WinCreateStdWindow (HWND_DESKTOP, WS_VISIBLE,
                  &flCreateFlags, szClientClass, " - Controls", 0L, NULL,
                  0, &hwndClient);

                                      /* message loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwndFrame);       /* destroy frame */
   WinDestroyMsgQueue(hmq);           /* destroy message queue */
   WinTerminate(hab);                 /* terminate PM usage */

   return 0;
   }

                                      /* client window procedure */
MRESULT EXPENTRY ClientWinProc (HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   HPS hps;
   RECTL rcl;

   ENTRYFDATA EFCtlData;
   static HWND hwndControl;           /* control window handle */
   static char pszText[MAXTEXT];      /* buffer for text */
   static SHORT cchText = 0;          /* length of text in szText */

   switch (msg)
      {
      case WM_CREATE:                 /* post private create msg */
         WinPostMsg(hwnd, CWPM_CREATE, NULL, NULL);
         return FALSE;                /* continue window creation */
         break;

      case CWPM_CREATE:               /* private create msg */

                                      /* init entry field control data */
         EFCtlData.cb = 8;
```

```
         EFCtlData.cchEditLimit = MAXTEXT - 1;
         EFCtlData.ichMinSel = 0;
         EFCtlData.ichMaxSel = MAXTEXT - 1;

         hwndControl = WinCreateWindow(
                          hwnd,
                          WC_ENTRYFIELD,    /* entry field window */
                          "Initial text",
                          WS_VISIBLE | ES_MARGIN,   /* with border */
                          10, 10, 175, 20,
                          hwnd, HWND_TOP, ID_WINDOW, &EFCtlData, NULL);

                                      /* entry field changed */
         WinPostMsg(hwnd, WM_CONTROL, MPFROM2SHORT(0, EN_CHANGE) , NULL);

                                      /* set keyboard focus to window */
         WinSetFocus(HWND_DESKTOP, hwndControl);
         break;

      case WM_CONTROL:                /* entry field change */
         if (SHORT2FROMMP(mp1) == EN_CHANGE)
            {
                                      /* get new entry field text */
            cchText = WinQueryWindowText(hwndControl, MAXTEXT, pszText);
                                      /* echo it */
            WinInvalidateRect(hwnd, NULL, FALSE);
            }
         break;

      case WM_PAINT:                  /* echo entry field to screen */
         hps = WinBeginPaint(hwnd, NULL, NULL);
         GpiErase(hps);
         rcl.xLeft = 10; rcl.xRight = 400;
         rcl.yBottom = 40 ; rcl.yTop = 60;
         WinDrawText (hps, cchText, pszText, &rcl, 0L, 0L,
            DT_LEFT | DT_VCENTER | DT_ERASERECT | DT_TEXTATTRS);
         WinEndPaint(hps);
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;

      default:                        /* default window processing */
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                       /* NULL / FALSE */
   }
```

```
/* ------- */
/* ENTRY.H */
/* ------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_WINDOW 1
#define CWPM_CREATE WM_USER
#define MAXTEXT 25
```

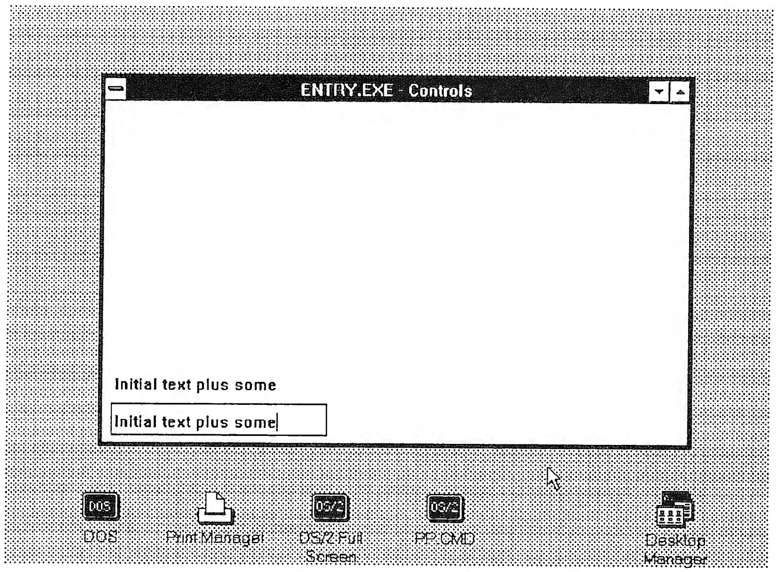


Figure 6-7. Output of ENTRY program

Creating Entry Fields with WinCreateWindow

Entry fields are control windows of the WC_ENTRYFIELD class. This class of window can have one of several styles, as specified by the *flStyle* argument of WinCreateWindow. Some of these are

Identifier	**Style**
ES_LEFT	Left justify text (default)
ES_CENTER	Center text
ES_RIGHT	Right justify text
ES_MARGIN	Draw rectangle around field

In our example we use ES_MARGIN to enclose the entry field in a box. We don't need to use ES_LEFT to left justify the text, since this is the default.

In previous examples the *pCtlData* argument to WinCreateWindow has been set to NULL. This is class-specific information, and for the entry field class this argument holds the address of a structure containing data about the entry field. The structure looks like this:

```
typedef struct _ENTRYFDATA  {
   USHORT cb;             /* Structure size (8 bytes) */
   USHORT cchEditLimit;   /* Maximum number of characters in entry field */
   USHORT ichMinSel;      /* Offset of first character selected */
   USHORT ichMaxSel;      /* Offset of last character selected */
}  ENTRYFDATA
```

The data in this structure is passed to the entry field window when it is created. The entry field window procedure receives a pointer to this structure in *mp1* of the WM_CREATE message. The entry field uses the information to construct itself. We set the maximum number of characters the entry field can hold to MAXTEXT (defined as 25 in ENTRY.H), and the selection range to the entire entry field.

Acting on Entry Field Changes

When the user makes a change in the entry field, such as adding or deleting characters, the entry field window updates the text in the window to reflect the user's action. It then transmits a WM_CONTROL message to its owner. The *mp1* parameter in this message is set to EN_CHANGE, a notification that the entry field has changed.

In our example the client window procedure also posts an EN_CHANGE notification in a WM_COMMAND message to itself, when it is first created. This ensures that the text from the edit field is initially copied to the screen when the program starts up, before the user has changed it.

When the client window procedure receives this notification, it executes WinQueryWindowText.

Get Text from Window

```
SHORT WinQueryWindowText(hwnd, cbBuf, pszBuf)
HWND hwnd        Handle of window from which text is copied
SHORT cbBuf      Length of buffer to receive text
PSZ pszBuf       Buffer to receive text

Returns: Length of text placed in buffer
```

This API reads the text string from the window specified and stores it in a buffer. The first argument is the handle of the window whose string will be copied. The second is the length of the buffer, and the third is the address

of the buffer. The function returns the length of the string stored in the buffer.

In the example, WinQueryWindowText is given the handle of the entry field control window, *hwndControl*. The text from the entry field is stored in *pszText*, and the length of the text is stored in *cchText* (the Hungarian notation *cch* indicates a count of characters).

Once we've extracted the text from the entry field and stored it in *pszText*, we want to display it on the client window. We could insert a section of code to do this in the *case* statement for WM_CONTROL, where it would be executed as soon as we learned what the text was with WinQueryWindowText. However, we also need to rewrite the text every time we receive a WM_PAINT message (when the user resizes the window, and so on). There's no point in having the same code in two places. It's preferable to cause the code in the WM_PAINT *case* statement to be executed whenever the text in the entry field changes. This can be done by having the code in the WM_CONTROL *case* statement generate a WM_PAINT message. The client window procedure will be sending a message to itself.

You might think you could simply transmit WM_PAINT directly to yourself, using WinPostMsg or WinSendMsg. However, this doesn't work. PM must generate the WM_PAINT message internally. To cause it to do this, we can execute the WinInvalidateRect function. This informs PM that a part of a window, in this case the client window, is invalid. When it learns the window is invalid, PM will transmit a WM_PAINT message, so that the window procedure can validate (paint) the window again.

Invalidate Window or Part of Window

```
BOOL WinInvalidateRect(hwnd, prcl, fIncludeChildren)
HWND hwnd                 Handle of window to be invalidated
PRECTL prcl               Rectangle to invalidate, NULL=entire window
BOOL fIncludeChildren     TRUE=Include descendants of window

Returns: TRUE if successful, FALSE if error
```

The second argument to this function specifies the rectangle to be invalidated. If this argument is NULL, the entire window is declared invalid. For simplicity, that's the value we use in this example.

Writing to the Screen

We discussed in Chapter 5 how to write text to the screen. We use the WinBeginPaint function to obtain the presentation space. We set the members of a structure of type RECTL to hold coordinates that will accommodate the text we want to write. Then we execute WinDrawText to do the writing.

Clipping to Children

The ENTRY program uses a CS_CLIPCHILDREN constant in the WinRegisterClass function in *main*. This specifies that the client window will clip itself to its children. Why do we need this constant?

The application rewrites the text displayed in the client window every time the user changes the text in the entry field. But before writing to the client window, the application needs to erase the old text. It uses the GpiErase function, which erases the entire client window.

Normally, anything drawn in a window is clipped to the window's parent, but not to any children the window may have. If you write in a window, you'll write over the child windows, and—since erasing is a form of drawing—if you erase a window you'll erase the child windows. Not worrying about clipping to child windows lets PM draw to a window more quickly. In our example, however, this means that erasing the client window will also erase the entry field window, which is its child. This isn't desirable, since we would then need to redraw the entry field window whenever the client window was redrawn.

The solution is to specify that the client window will clip itself to its children. The CS_CLIPCHILDREN constant does this. Now when the application erases the client window, the entry window is untouched.

CS_ and WS_

We've just described the use of the CS_CLIPCHILDREN identifier in the *flStyle* argument to WinRegisterClass. In the previous chapter we talked about WS_CLIPSIBLINGS. Why do we use a CS_ (class style) prefix in one case and a WS_ (window style) prefix in another?

You cannot modify the style of a predefined window class, such as a static window or other control. Its window procedure is inaccessible. But you can modify a particular *instance* of such a window. Thus you use a WS_ identifier in WinCreateWindow.

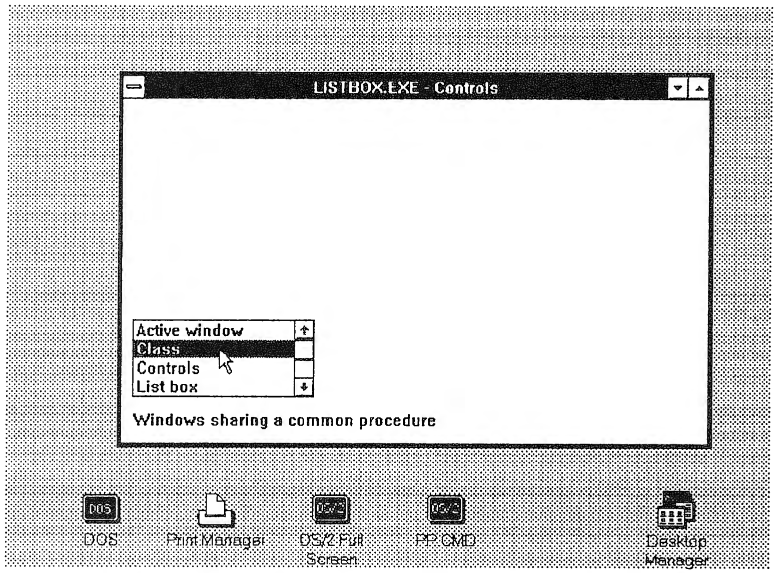


Figure 6-8. Output of LISTBOX program

On the other hand, you can modify the style of a window class you have defined yourself. Thus CS_ is used in WinRegisterClass in the ENTRY program to modify the behavior of the client window.

CHOOSING FROM A LIST BOX

A *list box* is a rectangle containing a list of items, usually short text strings. The user can select one of the items by clicking on it. The list box then notifies its owner that an item was selected, and the owner can take appropriate action. If there are too many items to fit in the box, the user can use the scroll bar on the right edge of the box to scroll additional items into view.

The list box in our example contains a list of words, such as "window" and "message." When the user selects a word, a brief definition is displayed below the list box, as shown in Figure 6-8.

Here is the client window procedure from LISTBOX.C, and LISTBOX.H.

```
                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc (HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   HPS hps;
   RECTL rcl;
   static SHORT ItemIndex;
   static HWND hwndControl;            /* control window handle */

   static CHAR *aszGlossary[2][NWORDS+1] = {
                   {"Active window", "Class", "Controls", "List box",
                    "Message", "Owner", "Parent", "Standard window",
                    "Window", ""},

                   {"Frame window ancestor of keyboard focus window",
                    "Windows sharing a common procedure",
                    "What this chapter is all about",
                    "The type of control window you are looking at",
                    "Information sent to a window",
                    "Window to which owner sends notification messages",
                    "Window to which child is clipped",
                    "Frame window plus children",
                    "An object",
                    "Select a word"}};

   switch(msg)
      {
      case WM_CREATE:
                                       /* post private create msg */
         WinPostMsg(hwnd, CWP_CREATE, NULL, NULL);
         return FALSE;                 /* continue window creation */
         break;

      case CWP_CREATE:                 /* private create msg */

         hwndControl = WinCreateWindow(    /* create a new window */
                          hwnd,
                          WC_LISTBOX,      /* list box control */
                          "Glossary", WS_VISIBLE, 10, 40, 150, 70,
                          hwnd, HWND_TOP, ID_WINDOW, NULL, NULL);

                                       /* add items to list */
         for (ItemIndex= 0; ItemIndex < NWORDS; ItemIndex++)
            WinSendMsg(hwndControl, LM_INSERTITEM, MPFROMSHORT(ItemIndex),
               aszGlossary[0][ItemIndex]);
         break;

      case WM_CONTROL:
         if (SHORT2FROMMP(mp1) == LN_SELECT)
            {
                                       /* query index of selected item */
            ItemIndex = SHORT1FROMMR(WinSendMsg(hwndControl,
                                       LM_QUERYSELECTION, NULL, NULL));
            WinInvalidateRect(hwnd, NULL, FALSE);   /* repaint */
            }
         break;

      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
         GpiErase(hps);
```

```
      rcl.xLeft = 10; rcl.xRight = 500;
      rcl.yBottom = 10 ; rcl.yTop = 30;
                                  /* write word definition */
      WinDrawText (hps, -1, aszGlossary[1][ItemIndex], &rcl, 0L, 0L,
         DT_LEFT | DT_VCENTER | DT_ERASERECT | DT_TEXTATTRS);
      WinEndPaint(hps);
      break;

   default:                       /* default window processing */
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }

return NULL;                      /* NULL / FALSE */
}
```

```
/* --------- */
/* LISTBOX.H */
/* --------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_WINDOW 1
#define CWP_CREATE WM_USER
#define NWORDS 9
```

The *main* function, and the Make and definition files, are similar to those in the ENTRY example.

Each word that appears in the list box, together with its definition, forms a pair of values in an array called *aszGlossary*. The index to this array, *ItemIndex*, is defined as a *static* variable so it will retain its value between calls to the client window procedure.

Creating List Boxes with WinCreateWindow

List boxes have the class WC_LISTBOX. WinCreateWindow is used with this argument to create the list box. The title is "Glossary," although this is not displayed on the screen. The box is located 10 pixels from the left and 40 pixels above the lower left corner of the frame window. It is 100 pixels wide and 50 pixels high, and has an ID of ID_WINDOW.

Placing Items in the List Box

In this program we place items into the list box from within the application. Putting the words into the list box involves sending a message, LM_INSERTITEM, to the list box. In this message *mp1* holds the index, or the place in the list where the item should go. We use the macro

MPFROMSHORT, defined in PMWIN.H, to convert the *ItemIndex* variable from SHORT to the MPARAM required by WinSendMsg. The item itself, which is a string such as "Window", is specified by *mp2*.

Instead of using an index to determine the ordering of the list items, we could have used the identifiers LIT_END, LIT_SORTASCENDING, or LIT_SORTDESCENDING to place the new item at the end of the list or to cause it to be inserted in forward or backward alphabetical order.

Acting on List Box Selections

When the user selects an item from the list box, the box sends its owner a WM_CONTROL message with the second half of *mp1* set to the LN_SELECT notification.

When our client window procedure gets this notification, it needs to find out which item from the list was selected. To do this, it sends an LM_QUERYSELECTION message to the list box. The reply to this message, conveyed by the return value from WinSendMsg, is the index of the selected item. In our example this value is assigned to *ItemIndex*.

Once the index of the selected item is known, WinInvalidateRect is used, as in the last example, to cause PM to generate a WM_PAINT message.

If the user double clicks on an item, or selects an item with a single click and then presses the ENTER key, the list box will send an LN_ENTER message to its owner. We don't make use of this in our example, but it's common for list boxes to be used in this way too.

Writing to the Screen

When it receives a WM_PAINT message, our client window procedure uses WinBeginPaint to obtain a cached micro presentation space. It then sets the coordinates of a rectangle in this space and writes the definition (the second item of the pair) from the *aszGlossary* array into this rectangle.

POSITIONING WITH A SCROLL BAR

While list boxes allow you to select a discrete item from a list, scroll bars permit a selection from a continuous range of values. Our example program, SCROLL, shows how to scroll a line of text that is wider than the screen. This is a common situation in word processors, spreadsheets, and other programs.

When you execute this program, you will see a standard window with a horizontal scroll bar at the bottom. Just above the scroll bar is a line of text. You can't see all of the line, since it is much longer than the window is wide. A typical view is shown in Figure 6-9.

To see the entire line of text, use the scroll bar to slide the text left and right. If you grab the slider with the mouse and move it back and forth, the text will move with it. The text scrolls smoothly and continuously, whether you move the slider slowly or quickly. Clicking on the gray areas on either side of the slider scrolls the text one screen-width left or right. Clicking on the arrows at the end of the scroll bar scrolls it one pixel. Programs can be written to scroll text pixel by pixel or in larger increments. In this example we use pixel scrolling, which is smoother and easier on the eyes.

Conceptuals

We'll discuss the concepts used in this program first and examine the listing later.

There are four major elements to consider: the window, the text string, a rectangle into which the text string will fit, and the scroll bar.

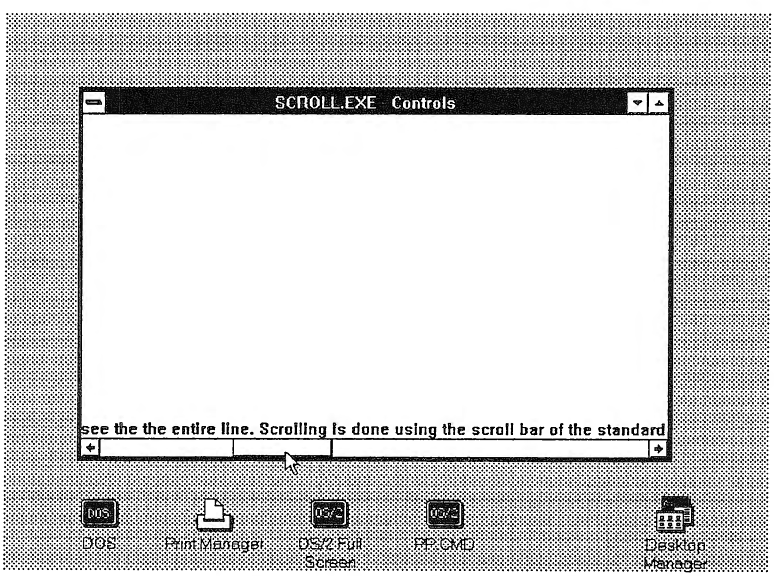


Figure 6-9. Output of SCROLL program

The Window

The window is a standard window. The user can resize it, so our application must be prepared to display different amounts of text, depending on the width of the window.

The Text String

The text string consists of characters that will be displayed in a proportional font. In a system with a monospaced font we could figure out the length of the text string by multiplying the character width by the number of characters, but in PM things aren't so simple. Fortunately, there is a function, GpiQueryTextBox, that can be used to calculate the actual dimensions of the string in pixels.

The Text Rectangle

A rectangle is created that is dimensioned, using the length and height from GpiQueryTextBox, to exactly hold the text string. When we display the text, using WinDrawText, we draw it into this rectangle.

The Scroll Bar

Confusion can arise with scroll bars because of differences in the perceptions of the user and the programmer.

Suppose the user is looking at the beginning (the left end) of the text line and wants to move toward the right end. This is achieved by clicking on the right scroll bar arrow or the right side of the gray area, or by moving the slider to the right. The user visualizes the window sliding over the text. By moving the slider to the right, the window is moved to the right, and text further toward the right is brought into view.

However, the window isn't really moving to the right. It's not moving at all; It's the *text* that's moving, and it's moving toward the *left*. So the program's response to rightward motion of the slider should be to move the text to the left. The slider and the text move in opposite directions.

Sliding Rectangles

Keep in mind that everything is measured *relative to the window.* Specifically, the lower left corner of the window is the origin. When we discuss the coordinates of the text rectangle, they're measured in this system. Think of

the window as fixed in the middle of the screen. Now imagine the text in its bounding rectangle. This rectangle is wider than the window, so some of it will stick out on the sides of the window. Since the window is fixed in our coordinate system, the text rectangle will have different coordinates, depending on where it's located relative to the window.

Let's give names to some of these points. The left side of the window is, by definition, always 0. The width of the window is *WindowLen.* The text rectangle is represented by a structure, *rclText,* and the left end of the text rectangle is *rclText.xLeft.* The length of the text rectangle is *TextLen.* Figure 6-10 shows the situation when the window is positioned in the middle of the text.

When the program first starts, the text is positioned so that the beginning of the first sentence is visible. At this point the left end of the text rectangle, *rclText.xLeft,* equals 0, as shown in Figure 6-11.

When the user wants to look at the right end (the last part) of the text, the right end of the text rectangle has the same X coordinate as the right edge of the window. At this point the *rclText.xLeft* variable equals *TextLen* minus *WindowLen.* This position is called *MaxLeft,* as shown in Figure 6-12.

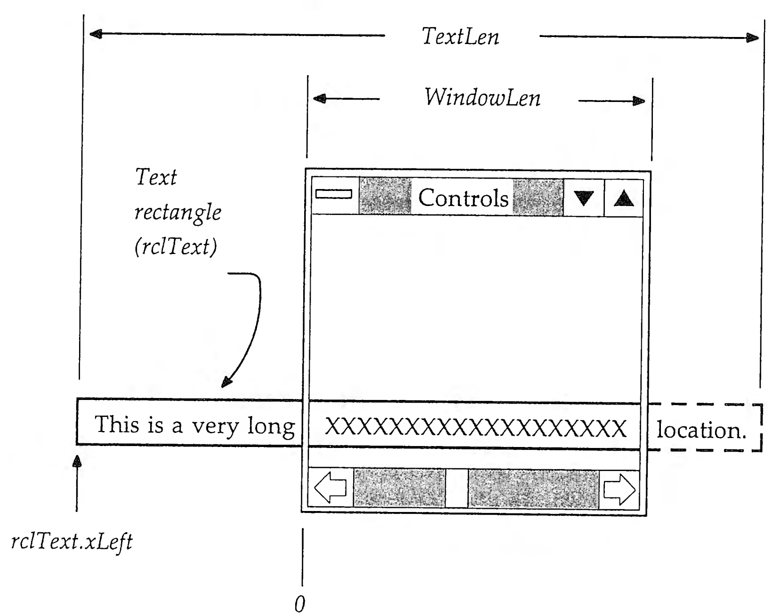


Figure 6-10. Window positioned in middle of text

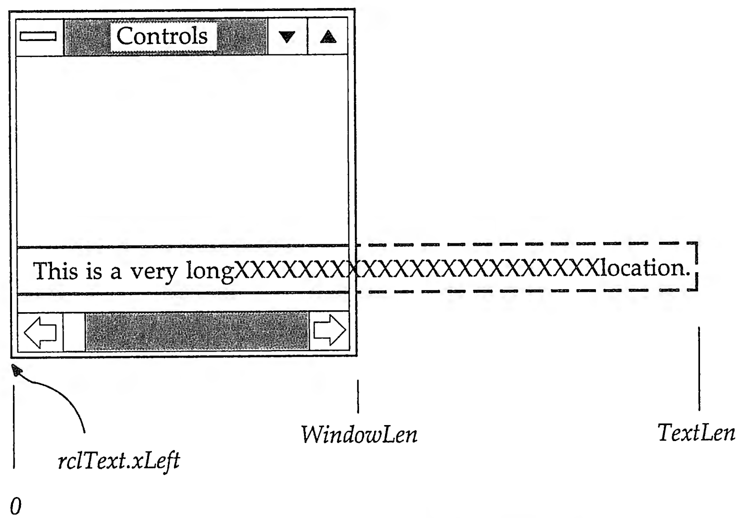


Figure 6-11. Window positioned at beginning of text

Text Protruding to Left of Window

When we display the text, we use WinDrawText to write to the client window. This takes as parameters the text string and a rectangle specifying where the text will be drawn. Typically, a part of this rectangle will be

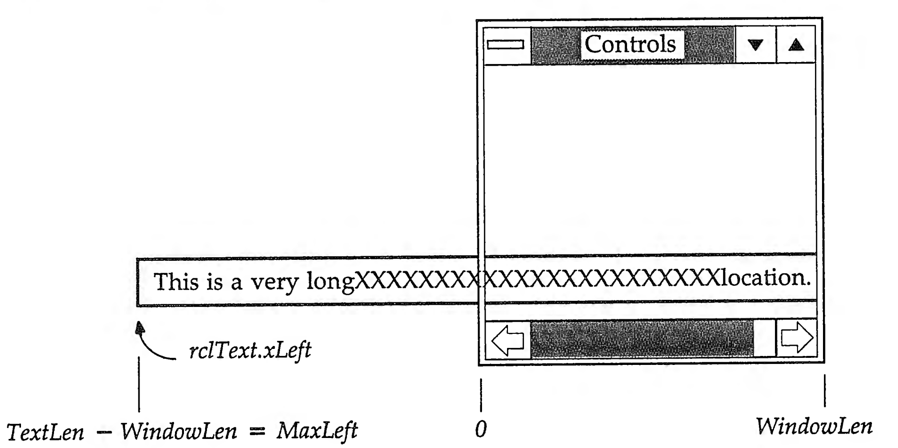


Figure 6-12. Window positioned at end of text

located to the left of the window. We don't want to display this text.

We could handle this by shortening the text rectangle so its left edge is on the left edge of the window. But if we did this, we would need to calculate where in the text string the left edge of the text rectangle falls. This would require calculating the length in pixels of the part of the text string to be removed. To find this, we would need to know the width in pixels of each character. The displayed text might well start in the middle of a character, which would require a different display technique. This would become rather complex.

However, we don't need to do this. We start writing at the beginning of the text, even if it's outside the window. Since a presentation space is clipped at the edges of its window, that part of the text that falls outside the window won't be visible.

Text Protruding to Right of Window

Text extending beyond the right side of the window is a different story. We may need the characters that come *before* those we want to display, so we can calculate where to place the displayed text. But we don't need the characters that *follow* the displayed text. Thus we can shorten the right side of the text rectangle so these characters are chopped off. Specifically, we can make the right side of the text rectangle correspond with the right side of the window. Text falling outside the text rectangle will never be written. Not writing this text improves efficiency, since the system makes all the calculations for text to be written, even if it's not actually displayed. Also, we don't need to change the right side of the text rectangle every time the text is moved; only the left side changes. Figure 6-13 shows the undisplayed text on the two sides of the window.

Slider Position

The slider position along the scroll bar is measured in arbitrary coordinates. Initially the range is from 0 on the left to 100 on the right. However, this range can be redefined by the program, using the SBM_SETSCROLLBAR message. It makes sense for the range of scroll bar positions to be the same as the range of positions of the text rectangle. The text rectangle can move from 0 to *MaxLeft*. However, because *MaxLeft* is a negative number, we reverse its sign when sending it to the scroll bar. The scroll bar range is thus from 0 to –*MaxLeft*.

There are really two sliders in the scroll bar. The first can be called the *temporary slider*. It appears as an outlined square and moves when the user drags the slider with the mouse. The second is the *permanent slider*. It is a

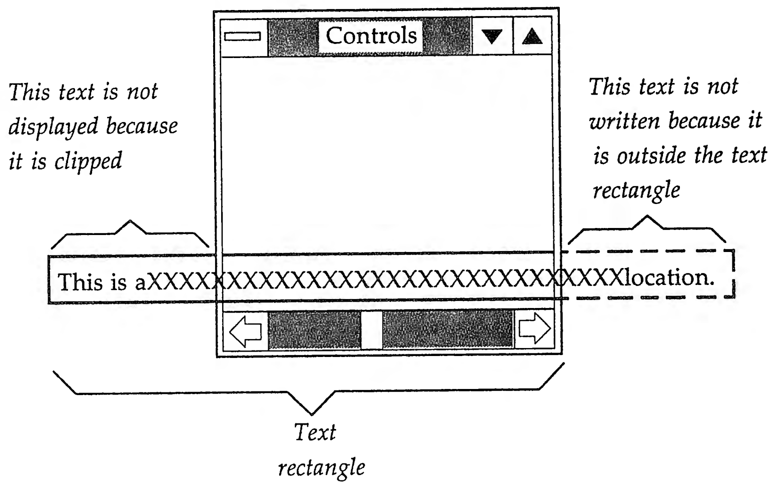


Figure 6-13. Undisplayed text

box with a solid outline and usually reflects the position of whatever is being scrolled; in this example it moves with the text. The scroll bar itself moves the temporary slider, but it's the application's responsibility to set the position of the permanent slider.

In the present example, the program moves the permanent slider so fast it's almost always in the same place as the temporary slider. To the user they appear as one, but the program must treat them separately.

Slider Size

Under OS/2 version 1.2, the program can control the size of the slider. This is useful to indicate to the user the proportion of the total field that is visible in the window. The bigger the slider, the more of the field is visible. In this example, the slider size depends on the length of the text string and the width of the window.

Now that you know what the program is supposed to do, let's look at the details.

Programming Details

This program requires some changes to the *main* function, so we show a complete new program, consisting of listings for SCROLL.C, SCROLL.H, SCROLL, and SCROLL.DEF.

```
/* -------------------------------- */
/* SCROLL.C - Using a scroll control */
/* -------------------------------- */

#define INCL_WIN
#define INCL_GPI
#include <os2.h>

#include <string.h>

#include "scroll.h"

USHORT cdecl main(void)
   {
   HAB hab;                          /* handle for anchor block */
   HMQ hmq;                          /* handle for message queue */
   QMSG qmsg;                        /* message queue element */
   HWND hwndFrame, hwndClient;       /* handles for windows */

                                   /* create flags for standard window */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_HORZSCROLL;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);        /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);  /* create message queue */

                                     /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                     /* create standard window */
   hwndFrame = WinCreateStdWindow (HWND_DESKTOP, WS_VISIBLE,
                  &flCreateFlags, szClientClass, " - Controls", 0L, NULL,
                  0, &hwndClient);

                                     /* message loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwndFrame);      /* destroy frame */
   WinDestroyMsgQueue(hmq);          /* destroy message queue */
   WinTerminate(hab);                /* terminate PM usage */

   return 0;
   }

                                     /* client window procedure */
MRESULT EXPENTRY ClientWinProc (HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   POINTL ptlText[TXTBOX_COUNT];
   HPS hps;
   USHORT fSliderMoved;
   static HWND hwndScrollBar;
   static RECTL rclText;
   static SHORT MaxLeft;
   static SHORT TextLen, WindowLen;
   static CHAR *szString = "This is a very long line of text that does \
not fit in the window, and so should be scrolled in order to see the \
```

```
entire line. Scrolling is done using the scroll bar of the standard \
window, which is an ownee of the frame, and so transmits its messages to \
the frame. The frame then transmits the scroll bar's messages to the \
client window procedure which scrolls the text by redisplaying it in the \
appropriate location.";

   switch(msg)
      {
      case WM_CREATE:
                                             /* get scroll bar handle */
         hwndScrollBar = WinWindowFromID(WinQueryWindow(hwnd, QW_PARENT,
                                                FALSE),FID_HORZSCROLL);
         hps = WinGetPS(hwnd);
         GpiQueryTextBox(                   /* get text box size */
            hps, (LONG)strlen(szString), szString, TXTBOX_COUNT, ptlText);
         WinReleasePS(hps);
         rclText.yBottom = 0;
         rclText.yTop = ptlText[TXTBOX_TOPLEFT].y -
                        ptlText[TXTBOX_BOTTOMLEFT].y;
         TextLen = (SHORT)(ptlText[TXTBOX_TOPRIGHT].x -
                           ptlText[TXTBOX_TOPLEFT].x);
         rclText.xLeft = 0;
         break;

      case WM_SIZE:
         WindowLen = SHORT1FROMMP(mp2);
         rclText.xRight = WindowLen - rclText.xLeft;
         MaxLeft = WindowLen - TextLen;
                                             /* set range & position */
         WinSendMsg (hwndScrollBar, SBM_SETSCROLLBAR,
            MPFROMSHORT(-rclText.xLeft), MPFROM2SHORT(0, -MaxLeft));
                                             /* set thumb size */
         WinSendMsg(hwndScrollBar, SBM_SETTHUMBSIZE,
            MPFROM2SHORT(WindowLen, TextLen), NULL);
         break;

      case WM_HSCROLL:
         fSliderMoved = TRUE;
         switch (SHORT2FROMMP(mp2))
            {
            case SB_LINERIGHT:
               rclText.xLeft--;
               break;

            case SB_PAGERIGHT:
               rclText.xLeft -= WindowLen;
               break;

            case SB_LINELEFT:
               rclText.xLeft++;
               break;

            case SB_PAGELEFT:
               rclText.xLeft += WindowLen;
               break;

            case SB_SLIDERTRACK:
               rclText.xLeft = -(SHORT)SHORT1FROMMP(mp2);
               break;
```

```
         default:
            fSliderMoved = FALSE;
            break;
         }
         if (fSliderMoved)
            {
                                    /* do not move out of range */
            if (rclText.xLeft > 0) rclText.xLeft = 0;
               else if (rclText.xLeft < MaxLeft)
                       rclText.xLeft = MaxLeft;
                                    /* set thumb position */
            WinSendMsg (hwndScrollBar, SBM_SETPOS,
               MPFROMSHORT(-rclText.xLeft), NULL);
                                    /* repaint window */
            WinInvalidateRect(hwnd, NULL, FALSE);
            }
         break;

      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
         WinDrawText(hps, -1, szString, &rclText, 0L, 0L,
            DT_LEFT | DT_VCENTER | DT_ERASERECT | DT_TEXTATTRS);
         WinEndPaint(hps);
         break;

      case WM_ERASEBACKGROUND:      /* erase frame's background */
         return TRUE;
         break;

      default:                      /* default window processing */
         return WinDefWindowProc(hwnd, msg, mp1, mp2);
         break;
      }

   return NULL;                     /* NULL / FALSE */
   }
```

```
/* -------- */
/* SCROLL.H */
/* -------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);
```

```
# ----------------
# SCROLL make file
# ----------------

scroll.obj: scroll.c
   cl -c -G2s -W3 -Zp scroll.c

scroll.exe: scroll.obj scroll.def
   link /NOD scroll,,NUL,os2 slibce,scroll
```

```
; ----------
; SCROLL.DEF
; ----------

NAME        SCROLL WINDOWAPI

DESCRIPTION 'Using the scroll bar control'

PROTMODE
STACKSIZE   4096
```

Let's go through SCROLL.C and consider the various new elements of the program in turn.

Creating Scroll Bars with WinCreateStdWindow

The scroll bar in SCROLL is not created with WinCreateWindow, as previous controls were in this chapter, but as a child of a frame window created with WinCreateStdWindow. The identifier FCF_HORZSCROLL in the *flCreateFlags* argument to this function causes the horizontal scroll bar to be added to the standard window.

Previous controls in this chapter were not only children of the client window, but ownees of this window as well. Thus notification messages traveled from the control directly to the client window procedure. The scroll bar in this example is an ownee, as well as a child, of the frame window, so messages travel from the scroll bar to the frame window. How do the messages get from the frame to the client window procedure, so we can process them? The frame forwards them automatically. Whenever a standard window is created with a client window, the frame window knows to forward appropriate messages it receives from its control, to the client. Figure 6-14 shows these two situations.

Accessing the Scroll Bar Window

We will be using messages to communicate with the scroll bar, so we need to know its window handle. Finding this handle is a two-step process. We know the handle of our client window: it's *hwnd,* which is transmitted as a parameter when our client window procedure is called. The WinQueryWindow API returns the handle of a window with a specific relationship to a known window. We use it to return the handle of the parent of our client window. This is the frame window. With this handle we use another

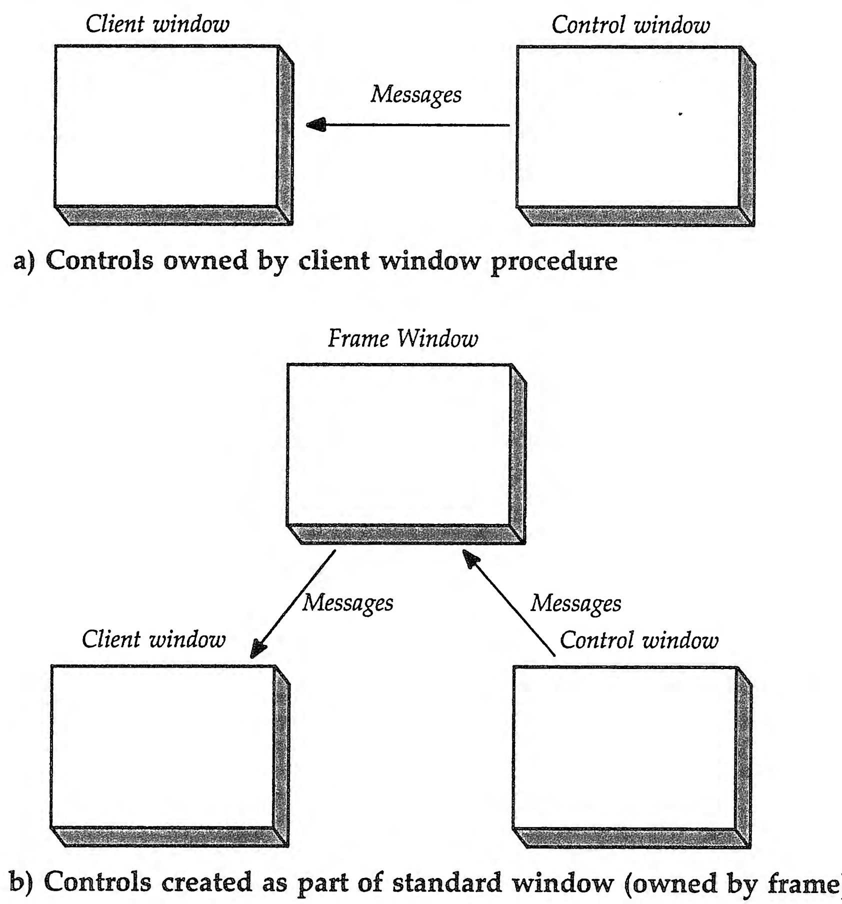


Figure 6-14. Messages from controls

function, WinWindowFromID, to obtain the handle of the scroll bar. In the listing, WinQueryWindow is nested inside WinWindowFromID. We execute these functions just after we receive the WM_CREATE message.

The WinQueryWindow Function

WinQueryWindow returns the handle of a window that has a specific relationship with the window used as an argument to the function. It answers questions like, "What's your parent's handle?"

Find Handle of Window Related to Given Window

```
HWND WinQueryWindow(hwnd, cmd, fLock)
HWND hwnd     Window handle
SHORT cmd     Relationship of desired window to hwnd
BOOL fLock    TRUE = lock related window; FALSE = no lock

Returns: Handle of window related to hwnd by relationship cmd
```

This function takes three arguments. The first is a window handle, which in our case is the client window handle. The second specifies the relationship of the desired window to the window specified. This should be a QW_ identifier (for Query Window), like QW_PARENT, QW_OWNER, QW_TOP, QW_BOTTOM, or QW_FRAMEOWNER. We want the parent of the client, so we use QW_PARENT. The third argument can be used to lock the window. We don't use it, so it's set to FALSE.

WinQueryWindow returns the handle of the frame window, which is the parent of the client window.

The WinWindowFromID Function

With the handle of the frame window we can now call WinWindowFromID to find the handle of the scroll bar.

Return Child Window Handle

```
HWND WinWindowFromID(hwndParent, id)
HWND hwndParent    Parent window handle
USHORT id          ID of child

Returns: Handle of child window of hwndParent, with ID number id
```

This function takes two arguments: the handle of the parent of the child window we want to find, and the ID of the child.

Every window has an ID number. We've seen examples of window IDs in previous programs: ID_BUTTON1 and ID_BUTTON2 in the BUTTONS program, for example. All the child windows of a frame window, that is, the control windows and the client window, are given fixed IDs by the system. These are defined in PMWIN.H:

ID of Control	Control
FID_CLIENT	Client window
FID_HORZSCROLL	Horizontal scroll bar
FID_VERTSCROLL	Vertical scroll bar
FID_MENU	Menu
FID_MINMAX	Minimize-maximize window
FID_SYSMENU	System menu
FID_TITLEBAR	Title bar

Given the handle of the frame window and the scroll bar ID FID_HORZSCROLL, WinWindowFromID returns the handle of the scroll bar.

Finding the Bounding Rectangle

We want to set up a rectangle that exactly surrounds the text. GpiQuery-TextBox serves the purpose. The use of a Gpi function requires that INCL_GPI be *#define*d, which we do in *main.* This function also requires the handle to our presentation space, which is returned by WinGetPS prior to calling GpiQueryTextBox.

Return Coordinates of Box Surrounding Text String

```
BOOL GpiQueryTextBox(hps, cchString, pchString, cpt, pptl)
HPS hps          Presentation space handle
LONG cchString   Number of characters in string
PCH pchString    Text string
LONG cptl        Number of points to return
PPOINTL paptl    Structure to contain points

Returns: GPI_OK if successful, GPI_ERROR if error
```

The first argument to this function is the handle of the presentation space obtained with WinGetPS. Second is the number of characters in the string, obtained from the *strlen* function. Third is the string itself. Fourth is an identifier that specifies the number of points to be returned by the function. TXTBOX_COUNT specifies the maximum information.

The fifth argument is the array into which the coordinates of the corners of the text rectangle will be placed. This is an array of type POINTL. POINTL is a structure defined in OS2DEF.H as

```
typedef struct _POINTL  {
  LONG x;       /* X coordinate of point */
  LONG y;       /* Y coordinate of point */
} POINTL;
```

The array indices are represented by the following identifiers:

Array Index Identifier	**Point**
TXTBOX_TOPLEFT	Top left
TXTBOX_BOTTOMLEFT	Bottom left
TXTBOX_TOPRIGHT	Top right
TXTBOX_BOTTOMRIGHT	Bottom right
TXTBOX_CONCAT	Concatenation point

Thus the X coordinate of the top left corner of the text rectangle is *ptlText[TXTBOX_TOPLEFT].x.*

We now define a rectangle, *rclText,* using dimensions derived from these coordinates. The lower left corner of *rclText* is initially set to 0,0, corresponding to the window positioned at the beginning of the text. The top of the rectangle is the height of the text. We also find the length of the text rectangle, *TextLen.*

Responding to Window Resizing

When the user changes the width of the window, our client window procedure receives a WM_SIZE message. When this happens, we want to recalculate *WindowLen.* The first half of *mp2* contains this new width, so we extract it with the SHORT1FROMMP macro and set *WindowLen* equal to it.

At this point we also set the *right* side of the text rectangle to this same value. This prevents the text rectangle from extending beyond the right side of the window. This allows us, as we noted earlier, to improve efficiency by not attempting to draw characters that would be clipped anyway.

Once we know the width of the window we can set the range of the scroll bar and the slider size. To do this we send SBM_SETSCROLLBAR and SBM_SETTHUMBSIZE messages to the scroll bar with WinSendMsg. (Other messages we can send the scroll bar include SBM_SETPOS, SBM_QUERYPOS, and SBM_QUERYRANGE.)

In SBM_SETSCROLLBAR, *mp1* sets the position of the slider. As we discussed, this is the negative of *rclText.xLeft.* We use the MPFROMSHORT macro from PMWIN.H to convert this to type MPARAM. The *mp2* parameter contains the range of the slider. The first half of *mp2* is the low end of

the range, which is 0, and the second half is the high end, which is the negative of *MaxLeft*. We use MPFROM2SHORT to convert these two values to type MPARAM.

In SBM_SETTHUMBSIZE the first half of *mp1* contains the number of currently visible elements. The second half of *mp1* contains the total number of elements. In the example the number of visible elements is the current window width in pixels (*WindowLen*), while the total number of elements is the length of the entire text string in pixels (*TextLen*). The *mp2* parameter is not used in this message and is set to NULL.

Responding to Scroll Bar Messages

When the user interacts with the horizontal scroll bar, it sends our client window procedure a WM_HSCROLL message. When this message is received, the second half of *mp2* specifies exactly what the user did. The possible values are

Identifier Received	User Action Generating Identifier
SB_LINELEFT	Click left arrow
SB_LINERIGHT	Click right arrow
SB_PAGELEFT	Click left gray area
SB_PAGERIGHT	Click right gray area
SB_SLIDERTRACK	Move slider
SB_SLIDERPOSITION	Move slider, release mouse button
SB_ENDSCROLL	Finish scrolling

The first half of *mp2* contains the position of the slider.

The *switch* statement in our example takes different actions depending on which SB_ identifier was received. All the messages we handle involve repositioning *rclText.xLeft,* the left end of the text rectangle. Clicking on the arrows moves it one pixel left or right, clicking on the gray areas moves it one window width, and moving the slider moves it to the position returned in *mp2.*

The program uses a flag called *fSliderMoved* to remember if the WM_HSCROLL message actually involved moving the slider. If it did, the program checks to be sure the rectangle has not moved too far left or right. The left end of the text can't go further right than the left edge of the window, and the right end of the text can't go further left than the right edge of the window. If the text rectangle has moved beyond these limits, it's set back to the limit. An SBM_SETPOS message is then sent to the

scroll bar to position the permanent slider at the new position. Finally, WinInvalidateRect is issued to cause PM to generate a WM_PAINT message so the text can be redrawn in its new location.

Writing the Text to the Screen

Once the position of the text rectangle *rclText* has been established, writing the text to the screen is relatively straightforward. This takes place whenever a WM_PAINT message is received.

Pixel Scrolling

In this example, we *rewrite* the text each time it is scrolled, even if it is only moved by one pixel. Another approach is to *scroll* the entire image pixel by pixel. The function WinScrollWindow can be used for this purpose. Using this function is a more efficient approach when an entire screenful of data must be scrolled, since the text need not be rewritten. However, in the present example it would have added substantially to the complexity of the program design. We'll demonstrate WinScrollWindow in Chapter 11.

7

RESOURCES

This chapter describes a new programming entity: the *resource.* Resources provide a standardized way to add information to your program. This information can consist of text, graphics, or any other kind of data in any format. We'll show what a resource is and discuss why you would want to use it. Then we'll show how resources are used to store various kinds of text and graphics data. Examples will cover string tables, icons, pointers, and programmer-defined resources.

Resources are important in programming menus and dialog boxes, to be described in the next two chapters. They are also used with fonts and bit maps.

WHAT ARE RESOURCES?

Program files in traditional architectures, such as MS-DOS .EXE files, contain two kinds of information: executable machine-language instructions, and the program's data.

In OS/2 a third kind of information is available to an application: resources. This information may be part of your program's .EXE file, or it may be placed in a DLL file. Once loaded into memory, resources, like a

program's normal code and data, occupy separate segments. Resources, however, occupy read-only data segments. Figure 7-1 shows the three kinds of data in .EXE files in MS-DOS and OS/2.

Unlike a program's normal code and data, a resource is not created from your program's source file. It starts off as a separate text file written by the programmer. This is then compiled into binary form using a utility called a *resource compiler.* This binary file is then added to the application's .EXE file (or to a DLL). The items in the resource are typically accessed from the application's code using specialized APIs such as WinLoadString, WinLoadPointer, and so on.

WHY USE RESOURCES?

A program's normal data is appropriate for small data items that are intimately associated with the executable code. However, more extensive

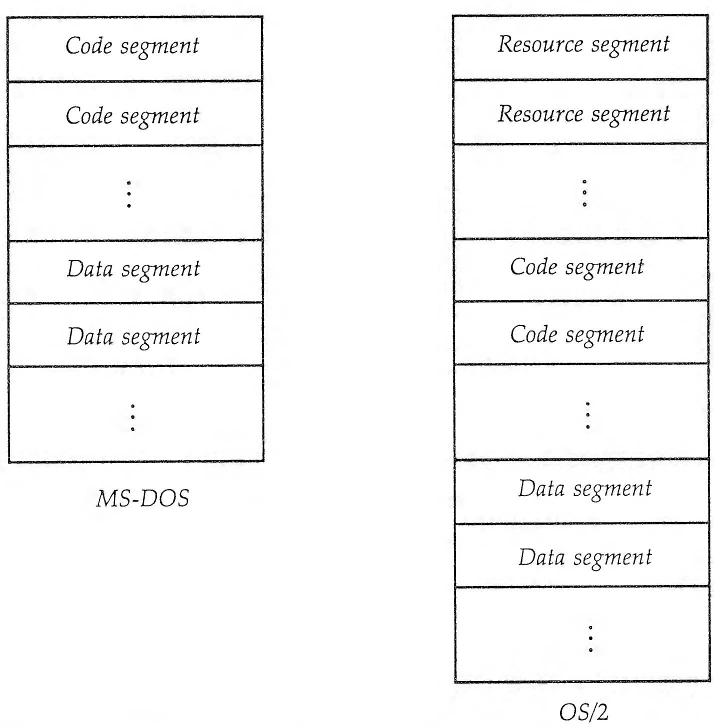


Figure 7-1. .EXE Files in MS-DOS and OS/2

kinds of data, such as text, graphics images, and other more esoteric data, present problems when stored as normal program variables. First, any change made in such data requires recompiling the source code file that contains it. Second, it may be difficult to create the data in the first place.

Resources help solve these problems. Since a resource is not part of a program's source code, changing it does not require recompilation of the source code file. And specialized tools are available for creating and maintaining many kinds of resources. Let's look at these points in more detail.

Independence from Source Files

In traditional systems, such as MS-DOS, most simple constant data items, such as text strings, are included in your program as normal C variables. Since it's in the same file as the executable code, such data is easily accessed by the application. The disadvantage of placing constant data in the source code file is that any change made to the data requires recompiling the entire file.

For instance, your application may contain static window text, dialog box text, menu items, and other text that is language specific. Suppose you want to adapt the application for use in a different country. You'll need to translate this text, edit the source code, and recompile, thus producing a new program. A different version of the source code will exist for each foreign language, making software maintenance more complicated.

One of the goals of OS/2 designers is to make it easy to adapt programs for use in other countries. Resources serve this goal. All language-specific data is placed in resources. To convert a PM application for a different language, only the resource file need be changed. The same executable files can be used without modification for all versions of the program. This simplifies program maintenance. It's even possible to ship a different version of the language-specific data to someone who does not have access to the source code, such as a software distributor in another country, since creating a version of the program for a new language requires only the executable code and the resource.

The same principle applies to other forms of customization. You can create different versions of a program for different clients, for example, without modifying or recompiling the source code.

Specialized Tools

Since the data in a resource is not embedded in a program's source code, it can be easily created and edited using special-purpose software tools. An

icon editor can create an icon, a dialog box editor can create a dialog box, and so on. In this chapter we'll see how the icon editor is used to create icons and pointers. In Chapter 9 we'll examine the dialog box. A font editor is supplied for creating new fonts. Tools for others kinds of data can be created as well: a music editor could be used to write a musical score to be displayed or played, and paint programs could create complex pictures in the form of resources.

Memory Efficiency

Resources occupy read-only segments in memory, and, like other kinds of read-only data, help to improve memory efficiency. Multiple instances of an application (or even different applications, if the resource is in a DLL file) can use the same copy of the resource in memory. Also, the resource can be overwritten when memory space becomes tight, and reloaded later.

Resources in DLLs

In many cases it is convenient to place resources in a dynamic link library (DLL) file rather than in an application's .EXE file. This permits the application to load the resource under program control, using the kernel API DosLoadModule. DLLs can be distributed separately from the .EXE file, allowing the use of a resource library, which can be used by many applications. In this chapter we'll show resources incorporated into an application's .EXE file.

CREATING RESOURCES

A resource starts out as a *resource script* file (or *resource definition* file; the names are interchangeable). This is a source file typed by the programmer, using an editor or word processor. It's analogous to a C source file.

For example, here's a resource file that creates one kind of resource, a *string table:*

```
#include <os2.h>
#include "example.h"

STRINGTABLE
   BEGIN
      ID_STRING1 "Warp factor 5, Mr. Sulu."
      ID_STRING2 "Klingons sighted in sector 7."
      ID_STRING3 "Fire phasers!"
   END
```

There are two kinds of lines in a resource script file: directives and statements.

Resource Directives

Directives look just like preprocessor directives in a C source code file. However, only a few directives are used in a resource script file. The most common are *#define* and *#include.* Flow-control statements like *#if* and *#else* can also be used, but we won't explore that possibility here. In the example, *#include* is used to include the files OS2.H and EXAMPLE.H in the resource. OS2.H contains definitions that are sometimes needed in the resource script file. The EXAMPLE.H file contains the definitions of ID_STRING1, ID_STRING2 and ID_STRING3, which appear later in the script file and in the program.

Resource Statements

Statements in a resource file specify the resource itself. Statements make use of *keywords,* which are used to specify particular types of resources and perform other functions. In the present example, the STRINGTABLE keyword specifies a string table resource. Other common resource-defining keywords are ICON, POINTER, MENU, DLGTEMPLATE, and FONT.

There are two categories of resource statements: single-line and multiline. The string resource shown here is an example of a multiline statement. Multiline statements are used to define resources within the resource script file itself. Single-line statements, which we'll encounter soon, are used to include resources that are located in different files.

Some resource-defining keywords may be followed by identifiers that specify options. For string tables, options can specify when the resource is loaded into memory and how it is managed once it's loaded. For example,

```
STRINGTABLE PRELOAD
```

causes OS/2 to load the STRINGTABLE into memory when the application is first loaded. Normally this resource is loaded only on execution of a specific API (for string tables, WinLoadString). Similarly, the FIXED option will keep the resource at a fixed location in memory. Normally the system will move or discard this resource when it needs more memory. The default values are usually more efficient, so in our examples we won't override them with special options.

Following the defining keyword are other statements that specify the contents of the resource. The keywords BEGIN and END are used to delimit the contents of the string table. If you prefer, you can use curly brackets (as in C) in place of BEGIN and END.

Each string in the table is specified in a separate statement. These statements consist of an ID number (for example, ID_STRING1) and a string of characters enclosed in double quotes. The ID numbers are used to refer to the resource in program statements and are typically defined in a header file. This makes them available to both the resource script file and the C source files.

Let's look at some programs that use resources.

A STRING TABLE EXAMPLE

Our first example uses a string table resource to hold two strings. One is used as the title of the program's window, and the other is written as text on the screen. Figure 7-2 shows the result when the program is executed.

Figure 7-2. Output of the STRING program

The number of files necessary to construct a program increases when we use resources. In this example there are five files: STRING.C, STRING.H, STRING, STRING.DEF, and the new STRING.RC.

```
/* --------------------------------- */
/* STRING.C - Using a string resource */
/* --------------------------------- */

#define INCL_WIN
#include <os2.h>

#include "string.h"

HAB hab;                                /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd, hwndClient;               /* handles for windows */
   char chBuf[257];                     /* buffer for strings */

                                        /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);     /* create message queue */

                                        /* load string */
   WinLoadString(hab, NULL, ID_TITLE, sizeof(chBuf), chBuf);

                                        /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                        /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
           szClientClass, chBuf, 0L, NULL, 0, &hwndClient);

                                        /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);              /* destroy frame window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */

   return 0;
   }

                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   HPS hps;
   RECTL rectl;
   static char szBuf[100];
```

```
   switch (msg)
      {
      {
      case WM_CREATE:
                                          /* load string resource */
         WinLoadString(hab, NULL, ID_MSG, sizeof(szBuf), szBuf);
         break;

      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
         WinQueryWindowRect (hwnd, &rectl);      /* get win dimensions */
                                                 /* display text */
         WinDrawText (hps, -1, szBuf, &rectl, 0L, 0L,
            DT_CENTER | DT_VCENTER | DT_ERASERECT | DT_TEXTATTRS);
         WinEndPaint(hps);
         break;

      default:                            /* default window processing */
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                           /* NULL / FALSE */
   }
```

```
/* -------- */
/* STRING.H */
/* -------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_TITLE 1
#define ID_MSG   2
```

```
# ----------------
# STRING make file
# ----------------

string.obj: string.c string.h
   cl -c -G2s -W3 -Zp string.c

string.res: string.rc string.h
   rc -r string.rc

string.exe: string.obj string.def
   link /NOD string,,NUL,os2 slibce,string
   rc string.res string.exe

string.exe: string.res
   rc string.res
```

```
; ----------
; STRING.DEF
; ----------
```

```
NAME        STRING WINDOWAPI
DESCRIPTION 'A string resource'

PROTMODE
STACKSIZE   4096
```

```
/* --------- */
/* STRING.RC */
/* --------- */

#include <os2.h>
#include "string.h"

STRINGTABLE
   BEGIN
      ID_TITLE " - This title is from a resource"
      ID_MSG "This message is also from a resource"
   END
```

You create the .RC file just as you would any other text file—by typing it in. To convert the .RC file to a binary form, you must compile it. The tool used for this is the resource compiler, RC. We'll describe what RC does, and then go back and see how the Make file for STRING controls the use of RC.

The Resource Compiler

The resource compiler actually performs two functions. First, it compiles a resource script file into binary form. That is, from the .RC file it generates a binary .RES file. Second, it adds this .RES file to the .EXE file created in the normal compile-and-link process. These two uses of RC can be combined into one, but it's more convenient to do them separately. We'll look at each of the two steps in turn. To use the resource compiler, the file RC.EXE must be available in the current directory or in the PATH.

Compiling the Resource Script File

To generate the machine-readable STRING.RES file from the resource script file STRING.RC, enter the following command:

```
rc -r string.rc
```

The *-r* switch tells the compiler not to add the compiled file, STRING.RES, to the .EXE file. Actually, the .RC extension, used in the statement shown,

doesn't need to be used; it's assumed. The compiled file, STRING.RES, also need not be named explicitly. By convention it has the same name as the .RC file and is given the .RES extension.

Combining the .RES and .EXE Files

To combine the .RES file with an application's .EXE file, invoke the resource compiler again:

```
rc string.res string.exe
```

This time the *-r* switch is not used, but the file extension .RES is. The executable file need not be entered if it has the same name as the .RES file. That is, you can also use

```
rc name1.res
```

Figure 7-3 shows the relationship of the resource compiler to the development process.

Why not combine these two uses of the resource compiler into one? In a typical development situation, especially with comparatively simple resources such as those used in this chapter, the resource will probably be compiled less often than the program. But the resource must be combined with the .EXE file every time the application is relinked. If the resource is recompiled every time it's combined with the .EXE file, as it is in the one-step approach, then it will be recompiled unnecessarily even if it has not been altered. We use the two-step process since it speeds program development in this situation.

The Make File

The Make file for STRING reflects this two-step use of the resource compiler. First (if it has changed), STRING.C is compiled into STRING.OBJ. Then (if it has changed) STRING.RC is compiled into STRING.RES, using the *-r* switch. The third step depends on STRING.OBJ: if it has changed, it's linked into STRING.EXE, and STRING.RES is combined with it to form the complete .EXE file. On the other hand, if the .OBJ file has not changed, but the .RES file is new, then the .RES file is combined with the .EXE file.

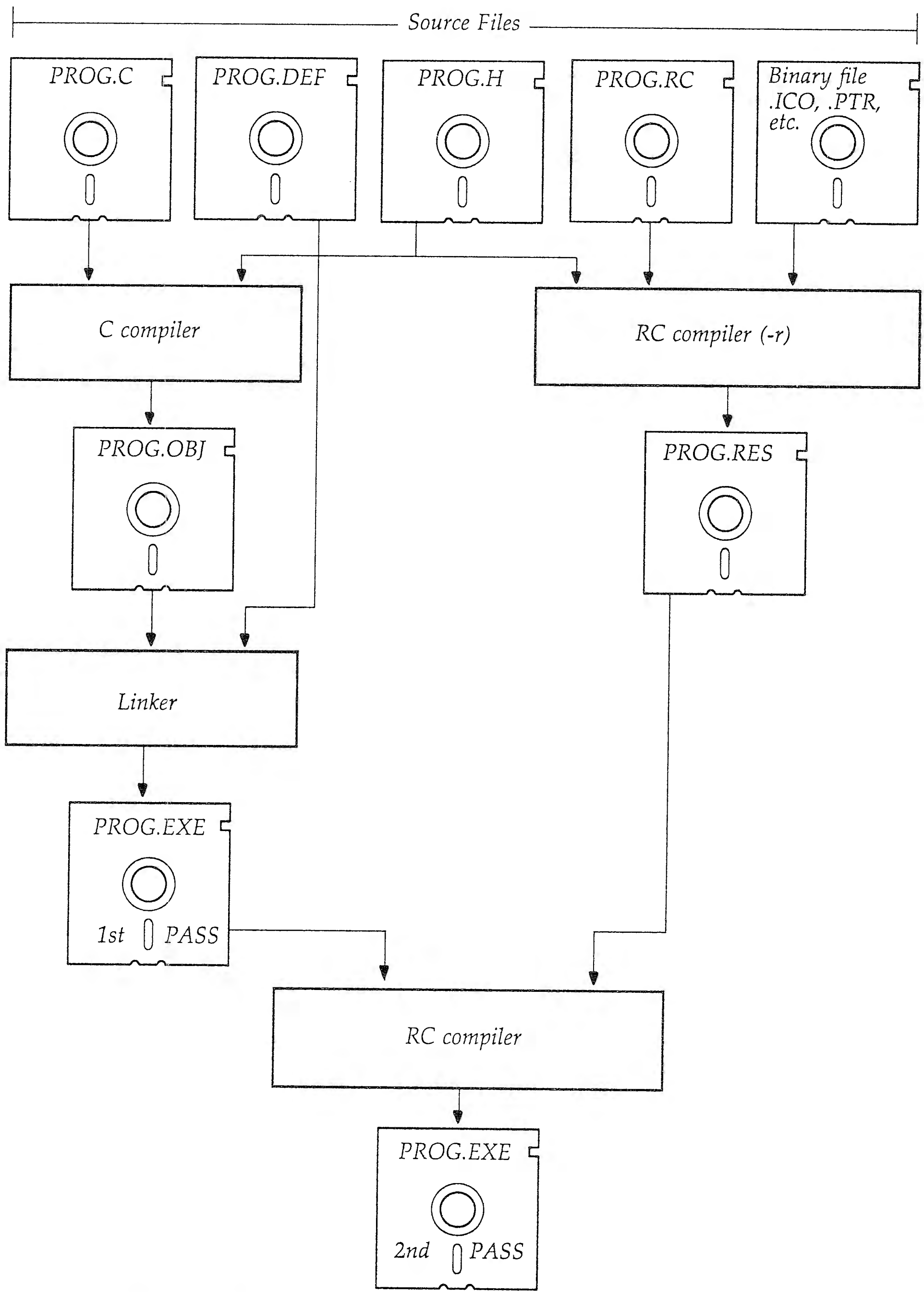


Figure 7-3. The resource compiler in program development

Loading Resources

Once a resource has been created and attached to its .EXE file, the application needs a way to access its contents. The method used depends on the resource and the use to which it will be put in the application. In the case of string resources we use the WinLoadString function.

Load a String from a Resource

```
SHORT WinLoadString(hab, hmod, id, cchMax, pszBuffer)
HAB hab          Anchor block handle
HMODULE hmod     Module handle (NULL if resource in .EXE file)
USHORT id        String ID
SHORT cchMax     Size of buffer to hold string
PSZ pszBuffer    Buffer to hold string

Returns: Length of string loaded (not counting terminating null),
         or 0 if error
```

The first argument to WinLoadString is the anchor block handle of the current thread, obtained from WinInitialize. The second is the resource module handle. This only applies if the resource was loaded dynamically from a DLL using DosLoadModule. In our example, the resource is located within our application's .EXE file, so this argument is set to null. Third is the string ID. This should be the same as that used in the resource script file. In our example these identifiers are defined in the .H file. Fourth is the size of the buffer that the string will be loaded into. A terminating null is added to the string when it's loaded, so the buffer should be at least one character larger than the size of the string. The fifth and last argument is the address of the buffer.

In the *main* part of the program, WinLoadString is used to load the string resource represented by ID_TITLE into the buffer *szBuf*. This variable is then used as the *pszTitle* argument to WinCreateStdWindow, so that the string " - This title is from a resource" appears in the window's title bar. WinLoadString returns the length of the string, but we don't make use of this, since a null-terminated string is assumed.

The second string, ID_MSG, is loaded into *szBuf* on receipt of the WM_CREATE message. Later, when the WM_PAINT message is received, the string loaded into *szBuf* is used as the *pchText* argument to WinDrawText. The string "This message is also from a resource" appears in the middle of the screen.

A string table can contain up to 16 strings, each of up to 255 characters.

ICONS

Resources can hold graphics images as well as text. One of the most common graphics resources is the *icon.* Icons are typically used to represent an application on the screen, both in the Group menu and when it's minimized. The user minimizes a window into its icon when the application is to continue running but occupy minimum screen space. The icon usually occupies a place near the bottom of the screen and, if double-clicked, expands back into the original top-level window. Single clicking on the icon brings up a minimized version of the application's System Menu.

Our example program creates a standard window and causes an icon to be associated with it. If you minimize the window, you'll see the icon appear at the bottom of the screen. If you double-click on the icon, the window reappears.

Here are the ICON.C, ICON.H, ICON, ICON.DEF, and ICON.RC files for this application.

```
/* ------------------------------- */
/* ICON.C - Using an icon resource */
/* ------------------------------- */

#define INCL_WIN
#include <os2.h>

#include "icon.h"

USHORT cdecl main(void)
   {
   HAB hab;                            /* handle for anchor block */
   HMQ hmq;                            /* handle for message queue */
   QMSG qmsg;                          /* message queue element */
   HWND hwnd;                          /* handle for frame window */

                                       /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_ICON;     /* icon included */

   hab=WinInitialize(NULL);            /* initialize PM usage */
   hmq=WinCreateMsgQueue(hab, 0);      /* create message queue */

                                       /* create standard window */
   hwnd=WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags, 0L,
           " - Minimize this window to see the icon", 0L, NULL,
           ID_FRAMERC,                 /* frame window resources ID */
           NULL);

                                       /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);             /* destroy frame window */
```

```
   WinDestroyMsgQueue(hmq);              /* destroy message queue */
   WinTerminate(hab);                    /* terminate PM usage */

   return 0;
   }
```

```
/* ------ */
/* ICON.H */
/* ------ */

USHORT cdecl main(void);

#define ID_FRAMERC 1
```

```
# --------------
# ICON make file
# --------------

icon.obj: icon.c icon.h
   cl -c -G2s -W3 -Zp icon.c

icon.res: icon.rc icon.h icon.ico
   rc -r icon.rc

icon.exe: icon.obj icon.def
   link /NOD icon,,NUL,os2 slibce,icon
   rc icon.res icon.exe

icon.exe: icon.res
   rc icon.res
```

```
; --------
; ICON.DEF
; --------

NAME         ICON   WINDOWAPI

DESCRIPTION 'An icon resource'

PROTMODE
STACKSIZE   4096
```

```
/* ------- */
/* ICON.RC */
/* ------- */

#include <os2.h>
#include "icon.h"

ICON ID_FRAMERC icon.ico
```

There's one file we can't show here, since it's a binary file. It's the file ICON.ICO that holds the image of the icon. To create this file we need to learn how to use a new utility.

Creating an Icon with ICONEDIT

Icons are most conveniently created using the ICONEDIT utility. This program is easy to use. Its top-level window contains a square area representing an icon. However, this area is much larger than an actual icon, which makes it easy to work with the individual pixels that constitute the image of the icon. In this area the mouse pointer assumes the shape of a pencil (or brush). Clicking a mouse button changes the color of the square, representing a pixel, which is at the pencil's tip. With this technique, which should be familiar to users of paint programs, you can quickly create your own icon.

Icon Resolution

The number of pixels used to display an icon on your screen is determined by the kind of display you have, as shown in the following table:

Display	Icon Size
CGA	32x16
EGA	32x32
VGA	32x32
8514	64x64

In ICONEDIT you can select the icon size you want to work with. The icon will be stored in this size, independent of the display adapter you happen to be using. When the icon is displayed, PM will convert it to the appropriate resolution.

Icon Color

Each pixel in the icon can be colored, transparent, or inverted. A colored pixel will be displayed in the color specified. A transparent pixel will be displayed in the underlying screen color. An inverted pixel will be displayed as the inverse of the underlying screen color. The pixel color is selected from the palette window supplied by the icon editor.

Icon Files

When you've drawn the icon to your satisfaction, use "Save As" from the "File" menu to save it as a file. You can give this file the .ICO extension. One possibility for an icon is shown in Figure 7-4.

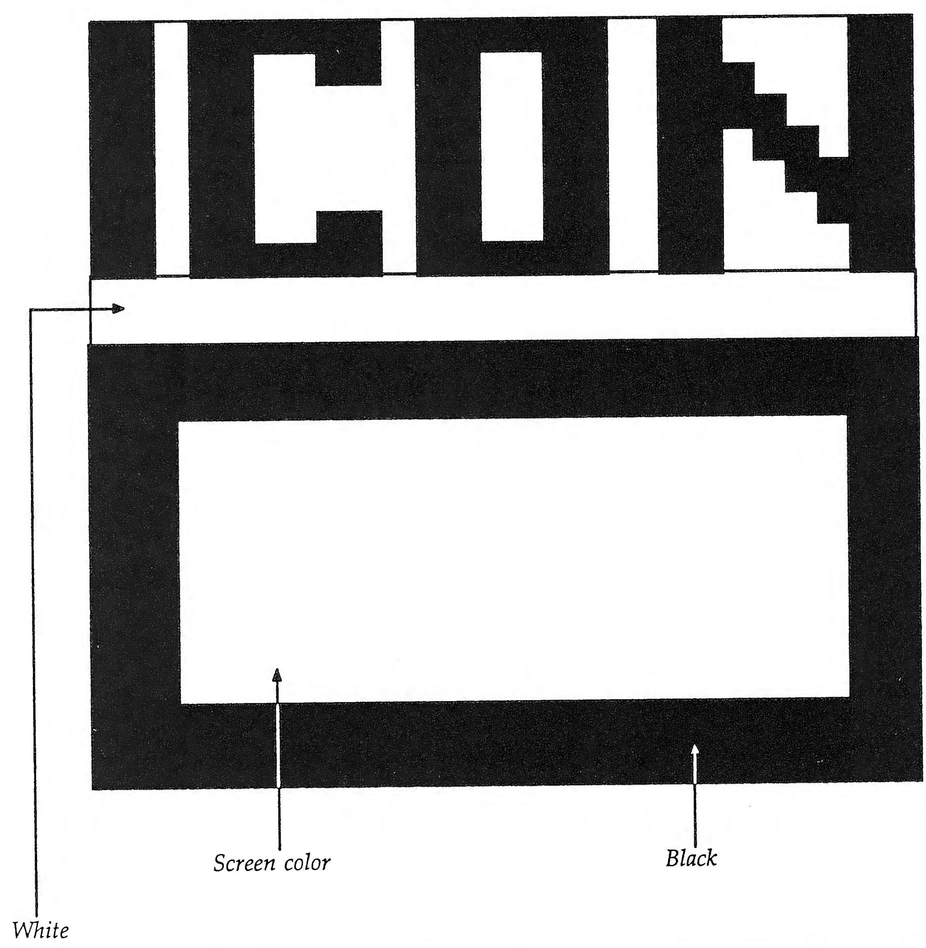


Figure 7-4. An icon

Icons in the Resource Script File

Since the resource script (.RC) file is a character file, and the .ICO file is a binary file, the contents of .ICO can't be placed directly in .RC. In our example the resource script file is very simple. It *#include*s the header file ICON.H, and then associates an ID number with the name of the file containing the icon. The statement is

```
ICON  ID_FRAMERC  ICON.ICO
```

This is an example of a single-line resource definition statement (as opposed to multiline statements, which are used, for example, in string resources). The ICON keyword specifies that it's an icon whose ID and file are being associated.

The Make File

In the Make file for ICON the compiled resource .RES file is dependent not only on the .RC script file, but also on the binary file containing the icon itself: ICON.ICO. This is important: whenever you change the binary .ICO file with ICONEDIT, the resource must be recompiled and recombined with the .EXE file. Changing the .ICO file does not by itself cause the icon in your program to change its appearance.

Loading an Icon

The WinCreateStdWindow function makes it easy to associate an icon with a standard window. You need to include the FCF_ICON identifier in the *flCreateFlags* argument and put the frame window resources ID, which is ID_FRAMERC in our example, in the *idResources* argument. We've used WinCreateStdWindow many times, but this is the first time we've used this argument.

Let's summarize the steps necessary to add an icon to a program:

1. Create a binary file containing the icon, using ICONEDIT.
2. Add an ICON resource definition statement to your resource file.
3. Add FCF_ICON to the *flCreateFlags* argument in WinCreateStdWindow.
4. Insert the icon's ID in the *idResources* argument in WinCreateStd-Window.

That's all you have to do. Our example program is so simple it can dispense with the client window procedure.

MOUSE POINTERS

A *mouse pointer* is the graphics figure, often an arrow, that moves on the screen when you move the mouse. Pointers are similar to icons. In fact, they are created with ICONEDIT just as icons are. They have the same dimensions and the same colors.

One difference between an icon and a pointer is that a pointer has a *hot spot.* This is the spot on a pointer that the system looks at to see where the pointer is pointing. It's the tip of an arrow, the center of cross hairs, or the end of a pointing finger. The hot spot is specified by selecting a menu option in ICONEDIT and clicking on the desired spot. The information on the hot spot's location is added to the pointer's binary file.

Our example program uses a user-written pointer to replace the system pointer whenever the pointer is inside the program's window. The new pointer is shown in Figure 7-5. The hot spot is the center of the cross hairs.

Here are the listings for the files POINTER.C, POINTER.H, POINTER, POINTER.DEF, and POINTER.RC.

```
/* ---------------------------------- */
/* POINTER.C - Use a pointer resource */
/* ---------------------------------- */

#define INCL_WIN
#include <os2.h>

#include "pointer.h"

USHORT cdecl main(void)
   {
   HAB hab;                           /* handle for anchor block */
   HMQ hmq;                           /* handle for message queue */
   QMSG qmsg;                         /* message queue element */
   HWND hwnd, hwndClient;             /* handles for windows */

                                      /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);         /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);   /* create message queue */

                                      /* register client window class */
```

```
    WinRegisterClass(hab, szClientClass, ClientWinProc, 0L, 0);

                                          /* create standard window */
    hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
              szClientClass, " - Pointer", 0L, NULL, 0, &hwndClient);

                                          /* messages dispatch loop */
    while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
       WinDispatchMsg(hab, &qmsg);

    WinDestroyWindow(hwnd);               /* destroy frame window */
    WinDestroyMsgQueue(hmq);              /* destroy message queue */
    WinTerminate(hab);                    /* terminate PM usage */

    return 0;
    }

                                          /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                      MPARAM mp2)
    {
    static HPOINTER hptr;

    switch (msg)
       {
       case WM_CREATE:
                                          /* load pointer */
          hptr = WinLoadPointer(HWND_DESKTOP, NULL, ID_POINTER);
          break;

       case WM_MOUSEMOVE:
                                          /* set pointer */
          WinSetPointer(HWND_DESKTOP, hptr);
          break;

       case WM_ERASEBACKGROUND:
          return TRUE;                    /* erase background */
          break;

       default:
          return(WinDefWindowProc(hwnd, msg, mp1, mp2));
          break;
       }

    return NULL;                          /* NULL / FALSE */
    }
```

```
/* --------- */
/* POINTER.H */
/* --------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_POINTER 1
```

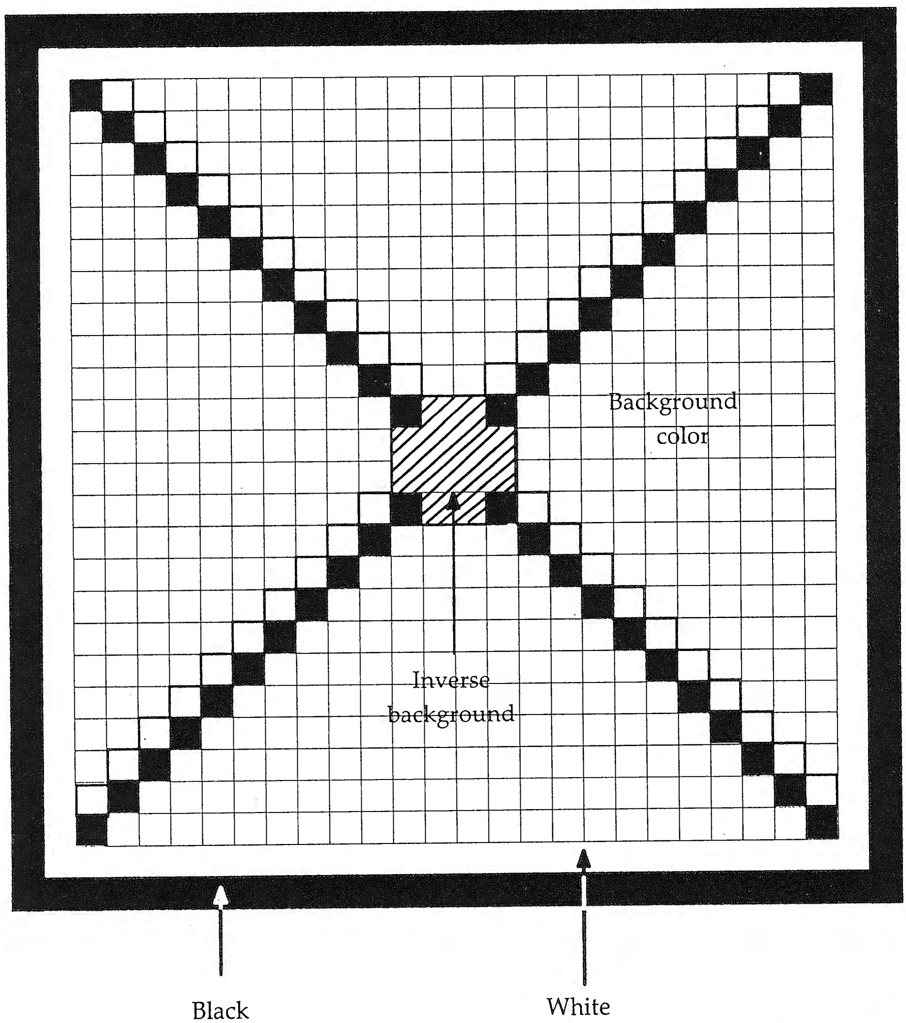


Figure 7-5. A custom mouse pointer

```
# ------------------
# POINTER Make file
# ------------------

pointer.obj: pointer.c pointer.h
   cl -c -G2s -W3 -Zp pointer.c

pointer.res: pointer.rc pointer.h pointer.ptr
   rc -r pointer.rc

pointer.exe: pointer.obj pointer.def
```

```
   link /NOD pointer,,NUL,os2 slibce,pointer
   rc pointer.res pointer.exe

pointer.exe: pointer.res
   rc pointer.res
```

```
; -----------
; POINTER.DEF
; -----------

NAME        POINTER WINDOWAPI

DESCRIPTION 'Use a pointer resource'

PROTMODE
STACKSIZE   4096
```

```
/* --------- */
/* POINTER.C */
/* --------- */

#include <os2.h>
#include "pointer.h"

POINTER ID_POINTER pointer.ptr
```

We can't show the binary file, POINTER.PTR, that holds the pointer itself. You can use ICONEDIT to create this file and save it with the .PTR extension.

The Resource Script File

The POINTER.RC resource script file is similar to that used for the icon in the last example. The line

```
POINTER  ID_POINTER  pointer.ptr
```

uses the keyword POINTER to associate an ID number, ID_POINTER, with the binary file POINTER.PTR, which contains the pointer.

Accessing the Pointer

Accessing a pointer is somewhat more complicated than was accessing the icon in the last example. There are two steps: loading it and setting it.

The WinLoadPointer function loads a pointer from a resource file into memory. The client window procedure in our example calls this function when it receives a WM_CREATE message.

Load Pointer from Resource

```
HPOINTER WinLoadPointer(hwndDesktop, hmod, idPtr)
HWND hwndDesktop   Desktop window handle (HWND_DESKTOP)
HMODULE hmod       Module handle (NULL if resource in .EXE file)
USHORT idPtr       Resource ID

Returns: Pointer handle, or NULL if error
```

Loading the pointer into memory doesn't mean that it appears on the screen. Another API, WinSetPointer, must be used to display it. This allows several pointers to be loaded into memory at the same time, and the appropriate one to be selected as needed. For instance, you could use WinSetPointer to display different pointers in different windows or in different areas of the same window.

Set Mouse Pointer

```
BOOL WinSetPointer(hwndDesktop, hptrNew)
HWND hwndDesktop          Desktop window handle(HWND_DESKTOP)
HPOINTER hptrNew          Pointer handle

Returns: TRUE if successful, FALSE if error
```

WinSetPointer must be executed whenever the mouse is moved, as signaled by the reception of a WM_MOUSEMOVE message.

You might think this function should be called only when the pointer started outside your window, where it has a different shape, and then moved inside. In this case, clearly it should be redrawn to the custom shape. On the other hand, if it moves entirely inside your window, it keeps its custom shape throughout and doesn't need to be redrawn. The problem is that there is no way for your application to tell whether a given mouse move originated outside the window or not. So the application must call WinSetPointer after every mouse move and let PM decide whether the pointer image needs to be changed. If no change is needed, then WinSetPointer has no effect.

With pointers, no special arguments need to be used in WinCreateStdWindow, as they are in creating icons. The pointer is associated with the client window procedure that loaded and set it, not with the standard window.

Here are the steps necessary to use a pointer:

1. Create a binary file containing the pointer, using ICONEDIT.

2. Add a POINTER definition statement to your resource file.
3. Load the pointer using WinLoadPointer.
4. Set the pointer with WinSetPointer whenever the mouse moves.

CUSTOM RESOURCES

A *custom* (or "programmer-defined") resource can be whatever you want: a text file of any length, or a binary file containing graphics or any other kind of information. Your application loads the data from the resource and interprets it according to its format.

Our example shows a programmer-defined resource used for a long text field. A string table can't be used in such a case, since it allows strings only up to 255 characters.

To show how several resources are combined in a resource file, the example uses an icon and also uses a string resource as the window title.

Following are the listings for CUSTOM.C, CUSTOM.H, CUSTOM, CUSTOM.DEF, CUSTOM.RC, and the text file CUSTOM.TXT, which constitutes the resource. You'll also need a binary file containing an icon, such as ICON.ICO from the earlier ICON example.

```
/* ---------------------------------------- */
/* CUSTOM.C - Using a custom defined resource */
/* ---------------------------------------- */

#define INCL_WIN
#define INCL_DOS
#include <os2.h>

#include "custom.h"

USHORT cdecl main(void)
   {
   HAB hab;                          /* handle for anchor block */
   HMQ hmq;                          /* handle for message queue */
   QMSG qmsg;                        /* message queue element */
   HWND hwnd, hwndClient;            /* handles for windows */
   char szBuf[257];                  /* buffer for strings */

                                     /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_ICON;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);        /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);  /* create message queue */
```

```
                                       /* load string */
  WinLoadString(hab, NULL, ID_TITLE, sizeof(szBuf), szBuf);

                                       /* register client window class */
  WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                       /* create standard window */
  hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
          szClientClass, szBuf, 0L, NULL, ID_FRAMERC, &hwndClient);

                                       /* messages dispatch loop */
  while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
     WinDispatchMsg(hab, &qmsg);

  WinDestroyWindow(hwnd);              /* destroy frame window */
  WinDestroyMsgQueue(hmq);             /* destroy message queue */
  WinTerminate(hab);                   /* terminate PM usage */

  return 0;
  }

                                       /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                         MPARAM mp2)
  {
  HPS hps;
  RECTL rectl;
  static SEL selURC;                   /* selector for user def. rc */

  switch (msg)
     {
     case WM_CREATE:
                                       /* get user resource (text) */
        DosGetResource(NULL, LONGSTRINGRC, ID_MSG, &selURC);
        break;

     case WM_PAINT:
        hps = WinBeginPaint(hwnd, NULL, NULL);
        WinQueryWindowRect (hwnd, &rectl);
        WinDrawText (hps, STRINGLEN, MAKEP(selURC, 0), &rectl, 0L, 0L,
           DT_CENTER | DT_VCENTER | DT_ERASERECT | DT_TEXTATTRS);
        WinEndPaint(hps);
        break;

     default:                          /* default window processing */
        return(WinDefWindowProc(hwnd, msg, mp1, mp2));
        break;
     }

  return NULL;                         /* NULL / FALSE */
  }
```

```
/* -------- */
/* CUSTOM.H */
/* -------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);
```

```
#define ID_FRAMERC 1
#define ID_TITLE 2
#define LONGSTRINGRC 1000
#define ID_MSG 101

#define STRINGLEN 271
```

```
# ----------------
# CUSTOM make file
# ----------------

custom.obj: custom.c custom.h
   cl -c -G2s -W3 -Zp custom.c

custom.res: custom.rc icon.ico custom.txt custom.h
   rc -r custom.rc

custom.exe: custom.obj custom.def
   link /NOD custom,,NUL,os2 slibce,custom
   rc custom.res custom.exe

custom.exe: custom.res
   rc custom.res
```

```
; ----------
; CUSTOM.DEF
; ----------

NAME        CUSTOM   WINDOWAPI

DESCRIPTION 'A custom defined resource'

PROTMODE
STACKSIZE   4096
```

```
/* --------- */
/* CUSTOM.RC */
/* --------- */

#include <os2.h>
#include "custom.h"

ICON ID_FRAMERC icon.ico

STRINGTABLE
   BEGIN
   ID_TITLE " - This title is from a resource"
   END

RESOURCE LONGSTRINGRC ID_MSG custom.txt
```

```
This is the text from the user defined resource. Since the text is longer
than 256 characters, it cannot be a string resource, and so a user defined
resource is used. A user defined resource does not have to be text however,
you can put anything you want in this resource.
```

The Resource Definition

The definition statement for the custom resource in the script file looks like this:

```
RESOURCE  LONGSTRINGRC  ID_MSG custom.txt
```

The RESOURCE keyword is followed by two ID numbers and the name of the file containing the resource. The first ID is the *type ID*. The resources we've used so far, such as icons and pointers, all have type IDs predefined by PM, using numbers from 0 to 255. Numbers from 256 to 65,535 are available for different types of custom resources. An application can use one or more type IDs. It might want to use one type ID for music files and another for picture data files, for example.

The second ID is the *resource ID*. This is the ID of a particular resource of a given type and is analogous to the IDs we've used for icons and other system-defined resources. These values can range from 0 to 65,535. A resource definition file may hold any number of RESOURCE statements, but two resources can't have the same combination of type ID and resource ID values.

The script file CUSTOM.RC also contains definitions for an icon and a string table. The necessary modifications to WinCreateStdWindow and the addition of WinLoadString generate the program window's icon and title, as described earlier in the STRING and ICON examples.

The DosGetResource Function

A kernel function, DosGetResource, retrieves the custom resource, either from the application's own .EXE file, or from a DLL. The contents of the resource are placed in a special read-only memory segment that this function allocates. In our example, DosGetResource is executed when the WM_CREATE message is received.

Get Resource from File, Load into Memory

```
USHORT DosGetResource(hmod, idType, idName, psel)
HMODULE hmod    Module (NULL if resource in .EXE file)
USHORT idType   Resource type ID
USHORT idName   Resource name ID
PSEL psel       Resource selector

Returns: 0 if successful, otherwise one of these error codes:
        ERROR_INVALID_MODULE
        ERROR_INVALID_SELECTOR
        ERROR_CANT_FIND_RESOURCE
```

The first argument to DosGetResource specifies the handle of a DLL module containing the resource. In this example we set it to NULL, since the resource is part of our application's .EXE file. The second argument is the type ID, and the third is the name ID. These values correspond to the type ID and name ID specified in the .RC file.

The fourth argument is the selector of the segment allocated for the resource. (Addresses in Intel 80x86 computers have two parts: a segment selector and an offset. The segment selector is the number placed in a segment register to access a particular segment, and the offset is the number of bytes into that segment.) The selector-and-offset address can be converted to a C pointer using the macro MAKEP (for Make Pointer).

The return value is in the style of all kernel functions: 0 if successful, otherwise one of a number of error codes. The error constants shown are defined in the BSEERR.H header file.

Accessing the Custom Resource

Different applications will do different things with the data from a custom resource. Our example is very simple: it displays the contents of the text file that constitutes the resource. To avoid complicating the program, we display the text in a single, long, horizontal line. To read the entire text, you'll need to repeatedly enlarge the program's window with the left and right sizing borders. You'll end up with a very wide window, part of which is shown in Figure 7-6.

The example uses WinDrawText to display the string from the resource when a WM_PAINT message is received. This is similar to the way WinDrawText displays string resources in the STRING example in this chapter. However, the macro MAKEP must be used in this function to convert the segment selector returned by DosGetResource to the pointer required by WinDrawText.

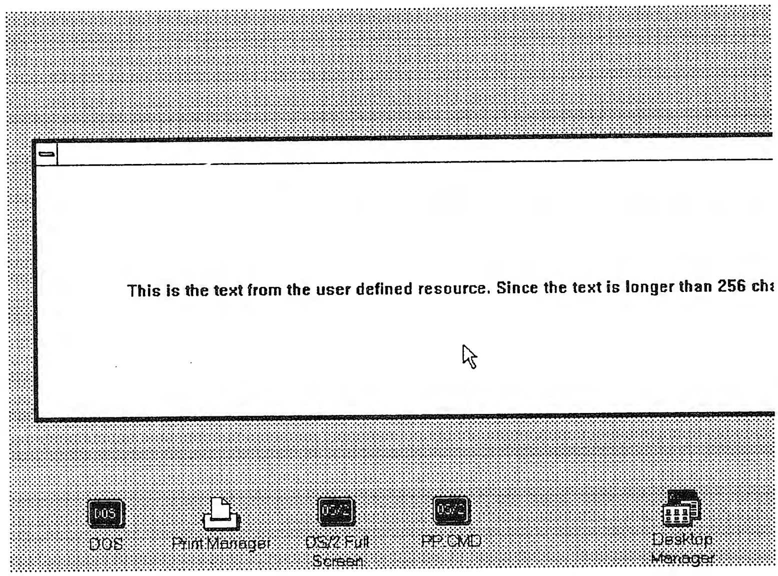


Figure 7-6. Output of the CUSTOM program

RESOURCES IN PM PROGRAMMING

So far we've examined fairly simple resources. Menus and dialog boxes, which we'll explore in the next two chapters, exploit resources more fully and will give you a better idea of their potential.

8

MENUS

Menus are a primary input medium in PM programs. Using resources, they are also easy for the programmer to create. This chapter explores menus, including such topics as submenus, checking and disabling menu items, and keyboard accelerators. We also introduce the subject of keyboard input, which ties in with accelerators; and timers, a useful tool in a variety of situations.

THE USER'S VIEW OF MENUS

Menus display—and permit the user to select—the main functions of the window in which they appear. They are typically used in the top-level window of an application to select the application's major functions. Menus offer the user an intuitively obvious way to instruct the program what to do. They are organized in a tree structure. Selecting an item from the top-level menu may cause an action, or it may cause a submenu to appear, with additional items. Some of these items may cause yet another level of submenu—a sub-submenu—to appear. By navigating through the menu tree to the appropriate item, the user can issue instructions of great complexity to an application, without remembering all the options or referring to a manual.

Menus, like the controls discussed in Chapter 6, are a type of control window: they notify their owners when something noteworthy happens, while handling routine tasks themselves. Menus are also similar to other kinds of controls from a user's viewpoint. As with push buttons, selecting menu items can cause actions to occur, and as with radio buttons, one of a list of options can be selected. Menu items can also be checked and unchecked like check boxes. However, menus are more fundamental than other kinds of controls. They are the primary way to offer choices to the user. Other types of controls are used in more specific circumstances and are often called up in response to a menu selection.

Let's examine the various elements of PM's menu system.

The Action Bar

The *action bar* (sometimes called the "menu bar") is a horizontal area of the standard window, located just below the title bar. Various short text phrases can appear in this area, like "File", "Options", and "Help". These items on the action bar constitute a window's *menu,* which is also called the "main menu" or "top-level menu."

To select an item from the top-level menu, the user positions the mouse pointer over the item's text and presses mouse button 1. This highlights the text and causes one of two things to happen. First, the item can cause a direct action. For example, help information will appear, or the program will execute some other task. Second, the item may invoke a submenu.

Submenus

A submenu is a list of items like those on the top-level menu. However, the items on a submenu are usually arranged vertically rather than horizontally. The submenu appears directly under the item in the top-level menu item that invoked it, as shown in Figure 8-1.

Submenus are often called "pull-down menus" or "drop-down menus" (or sometimes simply "drop-downs" or "pull-downs"). However, submenus can also be arranged horizontally, so the adjective "down" is potentially misleading. We'll use the term submenu. To select an item from the submenu, the user can release the mouse button while the item is highlighted, or simply click on the item. Again, an item on a submenu may cause a direct action, or it may cause yet another submenu—a sub-submenu—to appear.

Macintosh users may find this nomenclature confusing. On the Mac, the menu bar is not itself called a menu. First-level submenus are called simply "menus," and only the second and deeper levels of submenus are called "submenus."

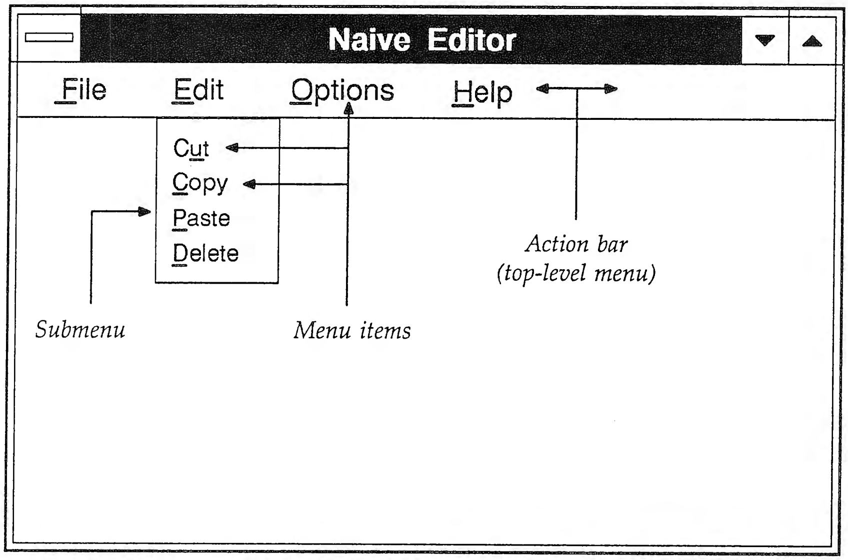


Figure 8-1. Action bar and submenu

Selections from the Keyboard

PM's philosophy is that all operations should be possible with the keyboard. Accordingly, you can select menu items entirely with keystrokes. This isn't as much fun as using the mouse, but some people don't have a mouse or find it inconvenient if their hands stray too far from the keys.

Selecting Menu Items with the Keyboard

Pressing F10 (or ALT) causes the first item on the action bar of the active window to be highlighted. The highlight can be moved across the action bar with the left and right arrow keys. Pressing ENTER selects the highlighted item. If this is a submenu, pressing the down arrow then moves the highlight through the items on the submenu, and pressing ENTER selects the highlighted item. The ESC key takes you up one level of submenus each time it's pressed.

Menu Mnemonics

If you're using the keyboard, and there are many items in a submenu, it may be too time-consuming to move the highlight to the desired item with

the down arrow. A *mnemonic* can be used to speed up the process. A mnemonic is an underlined character in a menu item. Once a menu or submenu has been selected, as just described, the user can select an item from it by pressing the keyboard key for the mnemonic character.

The mnemonic is usually the first character of the item, but if several items start with the same letter, a different letter may be used to avoid ambiguity. Figure 8-1 indicates menu mnemonics for several items. Mnemonics can be used for items in both the main menu and submenus.

Checking and Deactivating

Menu items can be *checked.* This means placing a small check mark to the left of the item name. Checking indicates that the state represented by an item is currently in effect. For instance, a font name shown in a menu that lists available fonts could be checked to show it is the font in use.

Menu items can also be *disabled.* A disabled item appears in half-tone (gray-looking) text. This indicates that it can't be selected. In a file-handling utility, for example, menu items specifying actions like "Print", "Delete", and "Move" might be disabled until a specific file has been selected for the items to act on.

Menu items are checked and disabled by the application. When the user selects certain items, the application checks them and unchecks others. The application can also enable or disable items on its own, in response to changing circumstances.

Keyboard Accelerators

Keyboard accelerators are keys or key combinations that cause a program action to be performed. They are usually used in conjunction with menus, so that typing an accelerator key and selecting a menu item perform the same task. The accelerator provides a shortcut for using the menu. Instead of selecting a submenu from the main menu, and then possibly a sub-submenu, and finally selecting the item, the user can simply press the accelerator key. For experienced users this may speed up the selection process. However, the key combination must be memorized, so accelerators are only appropriate when the user has acquired some experience with the application. They're easy to use, but not necessarily easy to learn.

So that users can learn the appropriate key combination for the accelerator, the keys are, by convention, displayed on the menu next to the corresponding item. The accelerator keys are tabbed to a separate column to separate them from the text of the menu item. Figure 8-2 shows several items with their accelerator key combinations indicated.

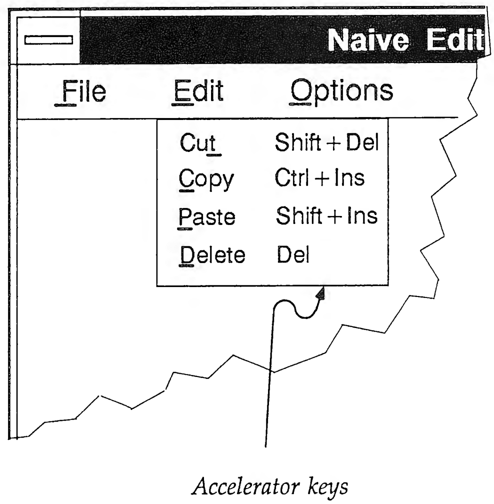


Figure 8-2. Keyboard accelerators

Don't confuse keyboard accelerators with mnemonics. An accelerator provides a quick way to select a particular menu item, no matter how deeply nested in submenus it may be. A mnemonic is a way to speed up the selection of an item as you work your way down through the menu tree using keystrokes instead of the mouse.

The System Menu

In previous chapters we mentioned the system menu icon, which appears on the left end of the title bar. Selecting this icon causes the system submenu to appear. The system submenu normally contains the items "Restore", "Move", "Size", "Minimize", "Close", and so on. These permit keyboard operation of these functions. This list can be modified by the application, but it's not good practice, since a standardized system menu is easier for the user.

The Help Menu

The SAA Common User Interface guidelines mandate that every application contain help text. Such text should be called up by pressing the F1 key, or by selecting the "Help" item from the action bar. This item should be the rightmost item on the action bar.

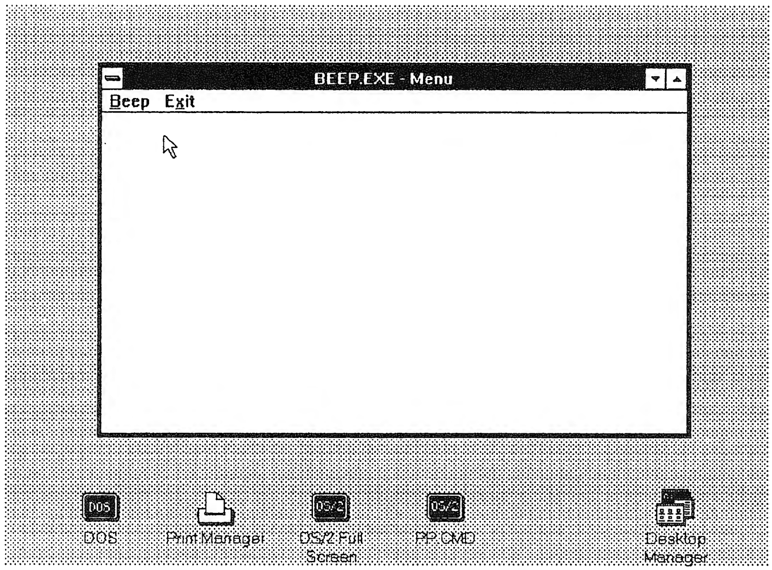


Figure 8-3. Output of the BEEP program

Now that we know what menus look like to the user, let's look at some programming examples.

ITEMS ON THE TOP-LEVEL MENU

Our first example is very simple. When the program is executed, two items appear on the menu bar: "Beep" and "Exit", as shown in Figure 8-3. Neither of these items invokes a submenu; they both take direct action. The first sounds a beep, and the second causes the program to terminate.

BEEP uses the files BEEP.C, BEEP.H, BEEP, BEEP.DEF, and BEEP.RC.

```
/* ------------------------ */
/* BEEP - Beep using a menu */
/* ------------------------ */

#define INCL_WIN
#include <os2.h>

#include "beep.h"

USHORT cdecl main(void)
   {
   HAB hab;                              /* handle for anchor block */
   HMQ hmq;                              /* handle for message queue */
   QMSG qmsg;                            /* message queue element */
   HWND hwnd, hwndClient;                /* handles for windows */
```

```
                                        /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_MENU;            /* menu included */

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);     /* create message queue */

                                        /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, 0L, 0);

                                        /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Menu", 0L, NULL,
             ID_FRAMERC,                /*   frame window resources ID */
             &hwndClient);

                                        /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);              /* destroy frame window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */

   return 0;
   }

                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   switch (msg)
      {
      case WM_COMMAND:
         switch SHORT1FROMMP(mp1)
            {
            case ID_BEEP:               /* Beep selected */
               WinAlarm(HWND_DESKTOP, WA_NOTE);
               break;

            case ID_EXIT:               /* Exit selected */
               WinPostMsg(hwnd, WM_QUIT, NULL, NULL);
               break;
            }
         break;

      case WM_ERASEBACKGROUND:
         return TRUE;                   /* erase backgound */
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                         /* NULL / FALSE */
   }
```

```
/* ------ */
/* BEEP.H */
/* ------ */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_BEEP 101
#define ID_EXIT 103
```

```
# --------------
# BEEP make file
# --------------

beep.obj: beep.c beep.h
   cl -c -G2s -W3 -Zp beep.c

beep.res: beep.rc beep.h
   rc -r beep.rc

beep.exe: beep.obj beep.def
   link /NOD beep,,NUL,os2 slibce,beep
   rc beep.res beep.exe

beep.exe: beep.res
   rc beep.res
```

```
; --------
; BEEP.DEF
; --------

NAME         BEEP    WINDOWAPI

DESCRIPTION 'Beep using a menu'

PROTMODE
STACKSIZE    4096
```

```
/* ------- */
/* BEEP.RC */
/* ------- */

#include "beep.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "~Beep", ID_BEEP
      MENUITEM "E~xit", ID_EXIT
   END
```

Resources and WinCreateStdWindow

As we noted, a menu is a type of control window, as are buttons, scroll bars, and so on. Like other controls, menus are most conveniently defined in a resource script file, and made part of a standard window.

Two changes must be made to WinCreateStdWindow to cause a menu resource to be added to a standard window. First, the FCF_MENU identifier is added to the *pflCreateFlags* argument. Second, the *idResource* argument is given the identifier of the menu resource. This same identifier (ID_FRAMERC, in the current example) is used for any resource, such as icons, that will be associated with the frame window.

The Resource File

The individual menu items, and indeed the structure of the entire menu tree, are defined in the resource script file. Refer to the BEEP.RC file for an example.

The MENU Keyword

The keyword MENU signifies the beginning of a menu resource. In the present example it's followed by the ID of the menu resource, ID_FRAMERC. (It can also be followed by the same options as other resources to control when it is loaded and how it's managed in memory.)

The MENUITEM Keyword

After the MENU statement comes a series of statements beginning with the keyword MENUITEM. These statements specify the actual entries that will appear in the menu on the action bar. This collection of MENUITEM statements is delimited by BEGIN and END keywords (or curly brackets). The order in which the items are arranged between the BEGIN and END statements is the order in which they will appear on the menu.

Each MENUITEM statement can contain up to five parts. First is the MENUITEM keyword itself. Second is the *item text.* This is the text that will appear in the menu on the screen. It's a string surrounded by quotes. In this string, a tilde (~) precedes the character that will automatically become the mnemonic for the menu item. This character will be displayed underlined.

Two other special characters may be used in menu item text. The '\t' character produces a tab, which moves text following it to another column, and '\a' causes the text following it to be right justified. We'll see an example of '\t' used with keyboard accelerators later in this chapter.

The third part of the MENUITEM statement is the *item ID* (sometimes called the *command value*). This is the value used to identify a particular menu item. In our example the item IDs are identifiers like ID_BEEP and ID_NOTESEL, defined in the BEEPS.H file. These IDs are normally used during message processing to determine which menu item was selected. They can also be used to refer to the menu item for other purposes, such as checking or deleting the item.

We don't use the fourth and fifth parts of the MENUITEM statement in BEEP, but we'll discuss them in the next example.

Message Processing

We've set up a menu resource and associated it with our standard window. Now, when we run the program, the menu items "Beep" and "Exit" will appear on the screen. What happens when the user attempts to select one of these items by clicking on it with the mouse (or selecting it from the keyboard)?

First, the *menu window procedure* will highlight the item. Our application doesn't need to worry about doing this; it's automatic. Note that, while there may be many menu items on the action bar and on submenus, all these items are controlled by a single menu window procedure (or *menu window*). The menu window procedure sends and receives messages concerning any of the individual menu items. The menu window procedure also takes other actions, such as highlighting an item when it is selected, dropping down submenus, and so on.

After it has highlighted a selected item, the menu window procedure's next step is to transmit a WM_COMMAND message to its owner, the frame window. The frame window will forward the message to our client window procedure, as it does with certain other messages from controls.

The first half of the *mp1* parameter of this message carries the ID of the menu item that was selected. By arranging these items in a *switch* statement, we can take the appropriate action for each item. In BEEP we sound the alarm when the command value is ID_BEEP and post a WM_QUIT message when the command value is ID_EXIT. (As we've seen in previous examples, posting the WM_QUIT message to itself causes the application to terminate.)

Summary of Menu Programming

Here are the steps necessary to install a menu in your standard window and process messages from it:

- Add a MENU definition to the resource file.
- Include FCF_MENU in the *flCreateFlags* argument to WinCreateStd-Window.
- Put the menu resource ID in the *idResource* argument to WinCreateStd-Window.
- Process WM_COMMAND: *mp1* contains the ID of the item selected.

SUBMENUS AND CHECKED ITEMS

In the last example, selecting items from the top-level menu on the action bar caused direct actions: beeping and exiting. More often, however, selecting items from the action bar causes a *submenu* to appear. The submenu typically drops down directly below the item on the action bar. In our next example, BEEPS (plural), there are the same two items on the menu bar: "Beep" and "Exit". However, when "Beep" is selected, instead of taking a direct action, it invokes a submenu with four items: "Sound beep", "Note", "Warning", and "Error", as shown in Figure 8-4.

Actually, there is a fifth item in this menu: The line under "Sound beep" is considered a separate menu item.

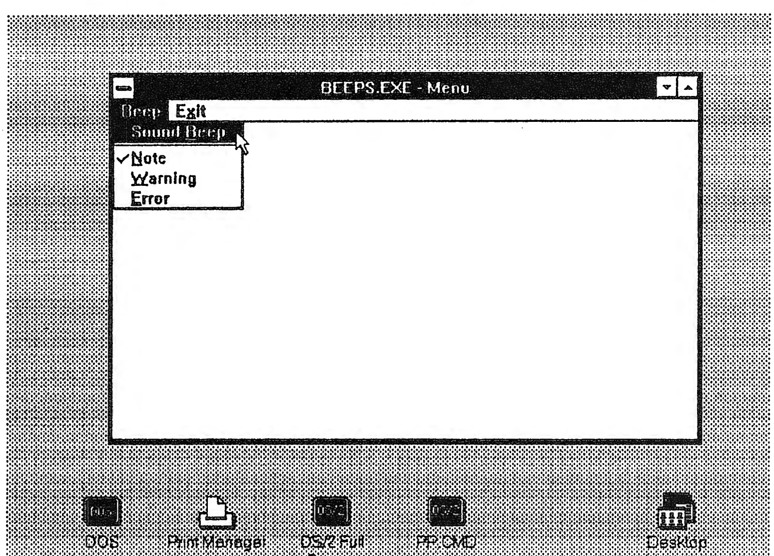


Figure 8-4. Output of the BEEPS program

Selecting "Sound beep" causes a note to sound. Selecting one of the items below the line causes the check mark to move to the item selected. When "Sound beep" is subsequently selected, different tones will occur, depending on the item checked. "Note" gives a higher tone than "Warning", which is higher than "Error". (These correspond to the three possible parameters to WinAlarm.) Selecting "Exit" from the action bar terminates the program.

Here are the listings for BEEPS.C, BEEPS.H, BEEPS, BEEPS.DEF, and BEEPS.RC.

```
/* ----------------------------------------------- */
/* BEEPS - Sound different beeps under menu control */
/* ----------------------------------------------- */

#define INCL_WIN
#include <os2.h>

#include "beeps.h"

USHORT cdecl main(void)
   {
   HAB hab;                             /* handle for anchor block */
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd, hwndClient;               /* handles for windows */

                                        /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_MENU;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);     /* create message queue */

                                        /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, 0L, 0);

                                        /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Menu", 0L, NULL, ID_FRAMERC, &hwndClient);

                                        /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);              /* destroy frame window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */

   return 0;
   }

                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
```

```
{
HWND static hwndMenu;               /* menu window handle */
static USHORT fsSound = WA_NOTE;    /* beep sound type */
static USHORT idCheckedItem = ID_NOTESEL;

switch (msg)
   {
   case WM_COMMAND:
      switch SHORT1FROMMP(mp1)
         {
         case ID_BEEP:              /* Beep selected */
            WinAlarm(HWND_DESKTOP, fsSound);
            break;
                                    /* beep sound type changed */
         case ID_NOTESEL:
         case ID_WARNINGSEL:
         case ID_ERRORSEL:
                                    /* un-check old sound type */
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
              MPFROM2SHORT(idCheckedItem, TRUE),
              MPFROM2SHORT(MIA_CHECKED, ~MIA_CHECKED));
                                    /* check new sound type */
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
               MPFROM2SHORT(SHORT1FROMMP(mp1), TRUE),
               MPFROM2SHORT(MIA_CHECKED, MIA_CHECKED));
            idCheckedItem = SHORT1FROMMP(mp1);

            switch SHORT1FROMMP(mp1)  /* change sound type */
               {
               case ID_NOTESEL: fsSound = WA_NOTE; break;
               case ID_WARNINGSEL: fsSound = WA_WARNING; break;
               case ID_ERRORSEL: fsSound = WA_ERROR; break;
               }
            break;

         case ID_EXIT:              /* Exit selected */
            WinPostMsg(hwnd, WM_QUIT, NULL, NULL);
            break;
         }
      break;

   case WM_CREATE:
                                    /* get menu window handle */
      hwndMenu = WinWindowFromID(WinQueryWindow(hwnd, QW_PARENT,
                                    FALSE), FID_MENU);
      break;

   case WM_ERASEBACKGROUND:
      return TRUE;                  /* erase background */
      break;

   default:
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }

return NULL;                        /* NULL / FALSE */
}
```

```
/* ------- */
/* BEEPS.H */
/* ------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_BEEP_SUBMENU 100
#define ID_BEEP 101
#define ID_NOTESEL 103
#define ID_WARNINGSEL 104
#define ID_ERRORSEL 105
#define ID_EXIT 110
```

```
# ---------------
# BEEPS Make file
# ---------------

beeps.obj: beeps.c beeps.h
   cl -c -G2s -W3 -Zp beeps.c

beeps.res: beeps.rc beeps.h
   rc -r beeps.rc

beeps.exe: beeps.obj beeps.def
   link /NOD beeps,,NUL,os2 slibce,beeps
   rc beeps.res beeps.exe

beeps.exe: beeps.res
   rc beeps.res
```

```
;----------
; BEEPS.DEF
; ---------

NAME         BEEPS   WINDOWAPI

DESCRIPTION 'Different beeps using a menu'

PROTMODE
STACKSIZE   4096
```

```
/* -------- */
/* BEEPS.RC */
/* -------- */

#include <os2.h>
```

```
#include "beeps.h"

MENU ID_FRAMERC
   BEGIN
      SUBMENU "~Beep", ID_BEEP_SUBMENU
         BEGIN
            MENUITEM "Sound ~beep", ID_BEEP
            MENUITEM SEPARATOR
            MENUITEM "~Note", ID_NOTESEL, , MIA_CHECKED
            MENUITEM "~Warning", ID_WARNINGSEL
            MENUITEM "~Error", ID_ERRORSEL
         END

      MENUITEM "E~xit", ID_EXIT
   END
```

The Resource File

The file BEEPS.RC is somewhat more complex than the resource file for BEEP. There are several new features.

Submenus

The new keyword in the resource file is SUBMENU. This is followed by an ID number for the submenu, although we don't make use of this ID in our application. The third field, *item-text,* is the text that will appear as the title of the submenu. SUBMENU introduces the list of menu items that will appear on the submenu. As in the top-level menu, this list consists of MENUITEM statements delimited by BEGIN and END.

Style and Attributes in MENUITEM

The fourth and fifth fields of the MENUITEM statement are optional: they're called the *menu item style* and the *menu item attribute.* If they exist, these fields are preceded by commas.

Some of the menu item styles are shown in the following table:

Menu Item Style	Effect
MIS_TEXT	*item-text* is a char string (default)
MIS_BITMAP	*item-text* is a bitmap resource identifier
MIS_SUBMENU	Item is a submenu
MIS_SEPARATOR	Item is a separator

MIS_BUTTONSEPARATOR	Button, last on line, follows separator
MIS_HELP	Item generates WM_HELP message
MIS_SYSCOMMAND	Item generates WM_SYSCOMMAND
MIS_OWNERDRAW	Item to be drawn by owner
MIS_STATIC	Item is nonselectable

Menu item attributes include

Menu Item Attribute	**Effect**
MIA_CHECKED	Places check mark next to item
MIA_DISABLED	Disables item so selecting it doesn't post message
MIA_FRAMED	Draws border around item
MIA_HILITED	Highlights item

In BEEPS we make use of only one attribute: MIA_CHECKED. This places an initial check mark beside the "Note" item in the "Beep" submenu.

The SEPARATOR Identifier

Note that one of the MENUITEM keywords is followed by the single identifier SEPARATOR. This draws a line on the menu, which can be used to separate different groups of items. In BEEPS it shows that "Sound beep" is not in quite the same category as "Note", "Warning", and "Error".

Processing Messages in BEEPS

The menu window procedure posts a WM_COMMAND message to the application when the user selects a menu item that causes an action, whether the item is on the top-level menu or a submenu. Note that WM_CONTROL is *not* sent when the user selects a submenu. Since selecting a submenu is an intermediate step in the user's search for the desired item, there's no use telling the application about it. (Applications that need more detailed control of the menu selection process may handle the WM_MENUSELECT message, which is generated by almost all menu activity, no matter how trivial.)

Using the item ID (the first half of *mp1* in the WM_COMMAND message), the application can figure out what item was selected. Selecting "Sound beep" from the "Beep" menu results in an item ID of ID_BEEP, which the application responds to by invoking WinAlarm. Depending on the value in the *fsSound* variable, which can be WA_NOTE, WA_WARNING, or WA_ERROR, this will sound one of three notes.

Selecting "Exit" causes a WM_COMMAND message with an item ID of ID_EXIT to be transmitted, which causes the WM_QUIT message to be posted and the program to terminate.

If the item selected is ID_NOTESEL, ID_WARNINGSEL, or ID_ERRORSEL, then the client window procedure must do two things. It must change the variable *fsSound*—the argument to WinAlarm—to the appropriate value (WA_WARNING, and so on). And it must move the check mark from the previously checked item to the newly checked item.

Checking and Unchecking

We've seen how messages are received by an application from the menu window procedure when an item is selected. To check or uncheck a menu item, the application must send a message back to the menu window procedure. This requires the application to know the handle of the menu window. Finding the handle involves a call to WinQueryWindow to find the handle of the parent of the menu window, which is the frame window. A call to WinWindowFromID then finds the handle of the menu window associated with that frame window. The result is placed in *hwndMenu*. We do this on receipt of a WM_CREATE message. (This technique was used in the SCROLL example in Chapter 6 to find the handle of a scroll bar, which is, like the menu window, a child of the frame window.)

Once the handle to the menu window is known, BEEPS executes WinSendMsg to send it a MM_SETITEMATTR message. This causes the attribute of the particular item to be changed. In BEEPS we change the attribute from MIA_CHECKED to ~MIA_CHECKED, and back again.

There are several dozen MM_ messages, which can be used to change various elements in menu items. We'll see other menu messages later.

The MM_SETITEMATTR Message

In MM_SETITEMATTR the *hwnd* argument is given the handle of the menu window. The *mp1* and *mp2* parameters are filled in as shown:

Parameter	Meaning
First half of *mp1*	Menu item ID
Second half of *mp1*	Search submenus? (TRUE=yes, FALSE=no)
First half of *mp2*	Mask of attributes to be changed
Second half of *mp2*	New attribute state

The first half of *mp1* is the ID of the menu item to be checked or unchecked, and the second half is set to TRUE if submenus should be searched for this menu item, and FALSE if only the top-level menu should be searched. These two halves of *mp1* are combined using the MPFROM2SHORT macro.

The attribute values that can be set using MM_SETITEMATTR are the same as those that can be applied to the item initially in the MENUITEM statement in the resource. These include MIA_CHECKED, MIA_DISABLED, and so forth. Several attributes can be ORed together.

Both halves of *mp2* are used to modify the attribute. The first half, *fsMask,* is a mask of the attributes to be changed; the second half, *fsData,* is the attributes themselves. The mask, *fsMask,* selects the bits to be changed, and *fsData* specifies the new values of the bits. Figure 8-5 shows how this looks.

Checking and Unchecking in BEEPS

To uncheck the old menu item, the client window procedure in BEEPS sends MM_SETITEMATTR with WinSendMsg. (The message should be sent, not posted, so we're assured the action is completed before continuing.) The menu item ID, which has been saved in *idCheckedItem,* and TRUE, which indicates we want to search in submenus, are combined using the MPFROM2SHORT macro to form the *mp1* value for the message MM_SETITEMATTR. MIA_CHECKED, and ~MIA_CHECKED (its complement), are combined with MPFROM2SHORT to form *mp2.*

To check the new menu item we need its ID. Since the main menu has just sent the WM_COMMAND message to tell us this item has been selected, the first half of the original *mp1* variable contains the ID number of the item, which can be extracted with SHORT1FROMMP. This is then combined with TRUE to form the *mp1* to be transmitted. MIA_CHECKED is combined with itself to form the *mp2.*

READING THE KEYBOARD

This section shows how to read the keyboard. We'll use keyboard keys to duplicate some of the menu selections in the BEEPS example. In our new example, BEEPSKBD, pressing the 'b' key causes the note to sound, and pressing CTRL-ALT-F10 causes the program to exit. Exploring keyboard input sets the groundwork for an understanding of keyboard accelerators, to be described in the next section.

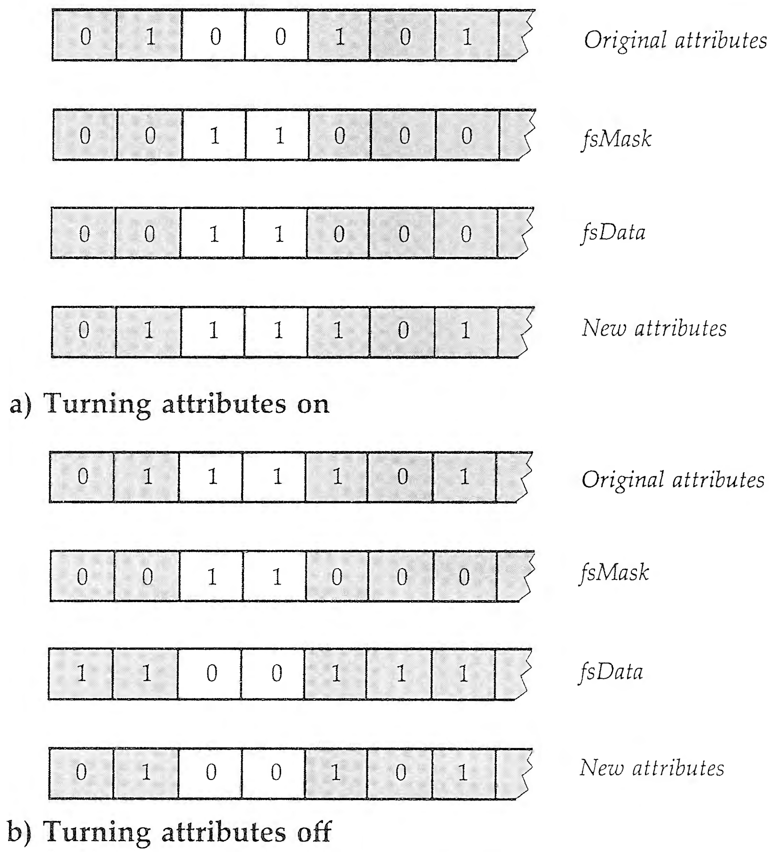


Figure 8-5. Attribute mask and data

Here is the listing for BEEPSKBD.C. All the other files, BEEPSKBD.H, BEEPSKBD, BEEPSKBD.DEF, and BEEPSKBD.RC, are similar to those for BEEPS.

```
/* ------------------------------------------------------------- */
/* BEEPSKBD - Sound different beeps under menu & keyboard control */
/* ------------------------------------------------------------- */

#define INCL_WIN
#include <os2.h>

#include "beepskbd.h"

USHORT cdecl main(void)
   {
```

```
   HAB hab;                              /* handle for anchor block */
   HMQ hmq;                              /* handle for message queue */
   QMSG qmsg;                            /* message queue element */
   HWND hwnd, hwndClient;                /* handles for windows */

                                         /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_MENU;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);            /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);      /* create message queue */

                                         /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, 0L, 0);

                                         /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Menu", 0L, NULL, ID_FRAMERC, &hwndClient);

                                         /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);               /* destroy frame window */
   WinDestroyMsgQueue(hmq);              /* destroy message queue */
   WinTerminate(hab);                    /* terminate PM usage */

   return 0;
   }

                                         /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   HWND static hwndMenu;                 /* menu window handle */
   static USHORT fsSound = WA_NOTE;      /* beep sound type */
   static USHORT idCheckedItem = ID_NOTESEL;

   switch (msg)
      {
      case WM_CHAR:                      /* a keystroke */
         if (!(SHORT1FROMMP(mp1) & KC_KEYUP))          /* key going down */
            {
            if (SHORT1FROMMP(mp1) & KC_VIRTUALKEY)     /* virtual key */
               {
               if (SHORT2FROMMP(mp2) == VK_F10         /* F10 */
                  && SHORT1FROMMP(mp1) & KC_ALT        /* and ALT */
                  && SHORT1FROMMP(mp1) & KC_CTRL       /* and CTRL */
                  && !(SHORT1FROMMP(mp1) & KC_SHIFT))  /* and not SHIFT */
                 WinPostMsg(hwnd, WM_QUIT, NULL, NULL);  /* Exit */
               }
               else if (SHORT1FROMMP(mp1) & KC_CHAR)   /* character key */
                       {
                       if ((SHORT1FROMMP(mp2) == 'B' ||  /* 'B' */
                           SHORT1FROMMP(mp2) == 'b')     /* or 'b' */
```

```
                                                /* and not SHIFT */
                    && !(SHORT1FROMMP(mp1) & KC_SHIFT ))
                   WinAlarm(HWND_DESKTOP, fsSound); /* Beep */
                }
         }
      break;

   case WM_COMMAND:
      switch SHORT1FROMMP(mp1)
         {
         case ID_BEEP:                /* Beep selected */
            WinAlarm(HWND_DESKTOP, fsSound);
            break;
                                      /* beep sound type changed */
         case ID_NOTESEL:
         case ID_WARNINGSEL:
         case ID_ERRORSEL:
                                      /* un-check old sound type */
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
               MPFROM2SHORT(idCheckedItem, TRUE),
               MPFROM2SHORT(MIA_CHECKED, ~MIA_CHECKED));
                                      /* check new sound type */
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
               MPFROM2SHORT(SHORT1FROMMP(mp1), TRUE),
               MPFROM2SHORT(MIA_CHECKED, MIA_CHECKED));
            idCheckedItem = SHORT1FROMMP(mp1);

            switch SHORT1FROMMP(mp1)  /* change sound type */
               {
               case ID_NOTESEL: fsSound = WA_NOTE; break;
               case ID_WARNINGSEL: fsSound = WA_WARNING; break;
               case ID_ERRORSEL: fsSound = WA_ERROR; break;
               }
            break;

         case ID_EXIT:                /* Exit selected */
            WinPostMsg(hwnd, WM_QUIT, NULL, NULL);
            break;
         }
      break;

   case WM_CREATE:
                                      /* get menu window handle */
      hwndMenu = WinWindowFromID(WinQueryWindow(hwnd, QW_PARENT,
                                      FALSE),FID_MENU);
      break;

   case WM_ERASEBACKGROUND:
      return TRUE;                    /* erase background */
      break;

   default:
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }

return NULL;                          /* NULL / FALSE */
}
```

This new program processes only one new message: WM_CHAR.

The WM_CHAR Message

WM_CHAR is the central (or should we say the key) message in keyboard input. It is posted every time a key is pressed or released, and stored in the queue of the window that has the *keyboard focus.*

Let's briefly examine the idea of keyboard focus. We've already seen that mouse messages usually go to the topmost window (in terms of the Z-axis) under the mouse pointer. Keystrokes, on the other hand, go to the window with the keyboard focus. A window is given the keyboard focus by an API, WinSetFocus, which is issued by PM or sometimes by an application. If no window is explicitly given the focus with WinSetFocus, then by default the active window has the focus. (For an example of using WinSetFocus, see the ENTRY example in Chapter 6.)

WM_CHAR Parameters

The parameters received with WM_CHAR provide details about the keystroke.

The *mp1* parameter has three fields. The first contains the scan code of the character. While scan codes are important in MS-DOS programming, they are not commonly used in PM. The second field is the repeat count. If a key is pressed more than once before the application removes it from its message queue, the messages are combined, and the repeat count indicates

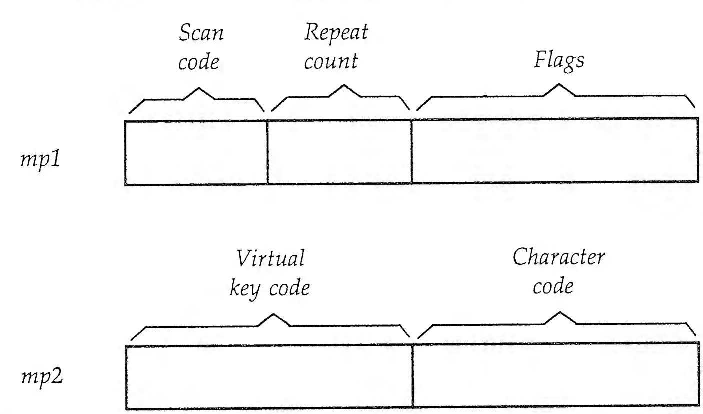


Figure 8-6. WM_CHAR parameters

the number of keystrokes represented by the message. The third field is a group of flags, which we'll examine in a moment.

The *mp2* parameter has two fields. The first is the *virtual key code.* PM uses this code to indicate physical keys, as opposed to ASCII character codes. The virtual key code is somewhat like the scan code, but is less machine-dependent. Non-character keys, like the function keys and the cursor control keys, are retrieved using this code. The second field is the normal character code (which is often the ASCII code). Figure 8-6 shows the WM_CHAR parameters.

The interpretation of these fields may be different in foreign languages.

WM_CHAR Flags

The flags in the third field of *mp1* contain information about the keystroke, the keyboard state, and about the other fields. Here are the meanings of the flags, which are defined in PMWIN.H:

Flag	Meaning
KC_CHAR	Valid value in character field (otherwise field is 0)
KC_SCANCODE	Valid value in scancode field (otherwise field is 0)
KC_VIRTUALKEY	Valid value in virtual key field (otherwise field is 0)
KC_KEYUP	Key was released (otherwise key was pressed)
KC_PREVDOWN	Key was previously down
KC_DEADKEY	Cursor should not move (for accents, etc.)
KC_COMPOSITE	Character formed from this key and preceding dead key
KC_INVALIDCOMP	Invalid combination with preceding dead key
KC_LONEKEY	Key not part of key combination
KC_SHIFT	SHIFT key was down when key pressed
KC_ALT	ALT key was down when key pressed
KC_CTRL	CTRL key was down when key pressed

Virtual Key Codes

The virtual key codes are represented by constants defined in PMWIN.H. Here are some major ones:

Virtual Key Code	Key
VK_BREAK	BREAK
VK_SHIFT	SHIFT
VK_CTRL	CTRL
VK_ALT	ALT

VK_CAPSLOCK	CAPSLOCK
VK_PAGEUP	PAGE UP
VK_PAGEDOWN	PAGE DOWN
VK_END	END
VK_HOME	HOME
VK_SCROLLLOCK	SCROLL LOCK
VK_NUMLOCK	NUMBER LOCK
VK_INSERT	INSERT
VK_DELETE	DELETE
VK_SYSRQ	SYSTEM REQUEST
VK_PRINTSCRN	PRINT SCREEN
VK_UP	Cursor control keys
VK_DOWN	
VK_LEFT	
VK_RIGHT	
VK_F1 to VK_F24	Function keys

Processing WM_CHAR Messages

When a WM_CHAR message arrives, the application's first task is to see if the keystroke was a key press or a key release. The KC_KEYUP flag is set on a key release, and not set on a key press. In general, we're only interested in key presses, so only if

```
SHORT1FROMMP(mp1) & KC_KEYUP
```

is FALSE do we continue processing the key. By eliminating key releases, we avoid further processing of half the WM_CHAR messages.

Next the flags typically are checked to determine if the key press is a virtual key code or a character code.

If the expression

```
SHORT1FROMMP(mp1) & KC_VIRTUALKEY
```

is TRUE, then the keystroke is a virtual key. The virtual key code can be extracted with

```
SHORT2FROMMP(mp2)
```

If the expression

```
SHORT1FROMMP(mp1) & KC_CHAR
```

is TRUE, then the keystroke is a character. The character code is then

```
SHORT1FROMMP(mp2)
```

In BEEPSKBD we're interested in recognizing only two keystrokes: the CTRL-ALT-F10 combination, and the 'b' key.

Once we ascertain that we're dealing with a virtual key, we check that the virtual key code is VK_F10, and that the KC_ALT and KC_CTRL flags are set. To exclude the SHIFT-CTRL-ALT-F10 combination (which in a more general program might conceivably be applied to a different function), we also require that the KC_SHIFT flag is not set. If these conditions are met, the program terminates.

If we're dealing with a character, there are two possibilities. If the caps-lock state is not active (the normal situation) we'll get a lowercase 'b'. If it is active, we'll get an uppercase 'B'. We want to sound the beep for both situations, in case the user is in caps-lock mode. On the other hand, if the SHIFT key is pressed, the user may have intended to select a different program function, so we make sure the KC_SHIFT flag is not set. If all these conditions are met, the program beeps.

Now that we know something about keystroke processing, let's look at another, simpler way to use keystrokes to select menu items.

KEYBOARD ACCELERATORS

A keyboard accelerator is a key or key combination that causes a WM_CONTROL (or sometimes a WM_HELP or WM_SYSCOMMAND) message to be sent. This message informs the application to take a particular action, in the same way that selecting a menu item tells the application to take an action.

Accelerators can be used by themselves, but they are typically used in connection with menus. There are two aspects to this. First, the program is arranged so that pressing the accelerator key combination sends the same message (and thereby causes the same program action) as making a particular menu selection. Second, the accelerator key combination is displayed next to the menu item, so that it's easy for the user to learn.

The next example program, BEEPSACC, is similar to BEEPS, but we've added accelerators to five menu items, as shown in this table:

Accelerator Keys	Menu Item
B	"Sound beep" (in "Beep" submenu)
ALT-N	"Note" (in "Beep" submenu)
ALT-W	"Warning" (in "Beep" submenu)
ALT-E	"Error" (in "Beep" submenu)
ALT-CTRL-F10	"Exit"

Keyboard Accelerator Tables

To implement keyboard accelerators, an *accelerator table* is added to the resource file. The ACCELTABLE statement defines the beginning of the table. The keyword ACCELTABLE is followed by the resource ID of the keyboard accelerator. (In our example this is ID_FRAMERC, as is the MENU keyword). Following this statement is a group of statements delimited by BEGIN and END statements (or curly brackets). Each statement associates a key or key combination with an ID number. If the same ID is used here that is used for a particular menu item, the accelerator keys will send the same message as the menu item and cause the same effect in the program.

There is no initial keyword in a statement in an accelerator table, as there is for menu items. For example, here's an accelerator table that associates the key 'a' with the ID number ID_ADDITION, and the F8 function key with the ID number ID_UNDERLINE:

```
ACCELTABLE ID_FRAMEWINDOW
   BEGIN
      "a"      ID_ADDITION
      VK_F8    ID_UNDERLINE, VIRTUALKEY
   END
```

This specifies that a WM_COMMAND message with a command value of ID_ADDITION is sent when the 'a' key is pressed. If this same ID is used in a menu item, then 'a' serves as an accelerator for that item.

Various options, separated by commas, may be placed after the item ID. In the example shown, the VIRTUALKEY option specifies that the constant VK_F8 is a virtual key code. The CHAR option specifies a character code, but it's the default, so we don't need to apply it to the 'a' character.

Options must also be used if a key will be used in combination with other keys. These options are SHIFT, CONTROL, and ALT. The following lines specify the letter 'a' used with the ALT key, and an 'X' typed with the SHIFT key depressed.

```
"a" ALPHA_ID, ALT
"X" CROSS_ID, SHIFT
```

Modifying the Menu Resource

It's standard practice to display the accelerator keys along with the menu item they invoke. This makes them easy to learn. The key or key combination is tabbed to the next column following the text of the menu item. This is accomplished by preceding the keys with the tab character '\t'. Typical items in a MENU resource that used the key combination ALT-E and 'X' as accelerators might look like this:

```
MENUITEM "Erase\tAlt+E", ID_ERASE
MENUITEM "Exit\tX", ID_EXIT
```

Of course it's up to the programmer to ensure that the key combination shown in the menu item is the same as that specified in the accelerator table.

Accelerators in BEEPSACC

Here are the listings for BEEPSACC.C, BEEPSACC.H, BEEPSACC, BEEPSACC.DEF, and BEEPSACC.RC.

```
/* ---------------------------------------------------------------- */
/* BEEPSACC - Sound beeps under menu & keyboard accelerator control */
/* ---------------------------------------------------------------- */

#define INCL_WIN
#include <os2.h>

#include "beepsacc.h"

USHORT cdecl main(void)
   {
   HAB hab;                              /* handle for anchor block */
   HMQ hmq;                              /* handle for message queue */
   QMSG qmsg;                            /* message queue element */
   HWND hwnd, hwndClient;                /* handles for windows */

                                         /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_MENU |
                         FCF_ACCELTABLE;         /* accelerator table */

   static CHAR szClientClass[] = "Client Window";
```

```
   hab = WinInitialize(NULL);            /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);      /* create message queue */

                                         /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, 0L, 0);

                                         /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
            szClientClass, " - Menu", 0L, NULL, ID_FRAMERC, &hwndClient);

                                         /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);               /* destroy frame window */
   WinDestroyMsgQueue(hmq);              /* destroy message queue */
   WinTerminate(hab);                    /* terminate PM usage */

   return 0;
   }

                                         /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                        MPARAM mp2)
   {
   HWND static hwndMenu;                 /* menu window handle */
   static USHORT fsSound = WA_NOTE;      /* beep sound type */
   static USHORT idCheckedItem = ID_NOTESEL;
   HPS hps;
   RECTL rcl;

   switch (msg)
      {
      case WM_HELP:                      /* "Help" requested */
         hps = WinGetPS(hwnd);
         WinQueryWindowRect (hwnd, &rcl);
         WinDrawText (hps, -1, "Not much help, is it?!", &rcl,
            0L, 0L, DT_CENTER | DT_VCENTER | DT_ERASERECT | DT_TEXTATTRS);
         WinReleasePS(hps);
         break;

      case WM_COMMAND:
         switch SHORT1FROMMP(mp1)
            {
            case ID_BEEP:                /* "Beep" selected */
               WinAlarm(HWND_DESKTOP, fsSound);
               break;
                                         /* beep sound type changed */
            case ID_NOTESEL:
            case ID_WARNINGSEL:
            case ID_ERRORSEL:
                                         /* un-check old sound type */
               WinSendMsg(hwndMenu, MM_SETITEMATTR,
                  MPFROM2SHORT(idCheckedItem, TRUE),
                  MPFROM2SHORT(MIA_CHECKED, ~MIA_CHECKED));
                                         /* check new sound type */
               WinSendMsg(hwndMenu, MM_SETITEMATTR,
                  MPFROM2SHORT(SHORT1FROMMP(mp1), TRUE),
                  MPFROM2SHORT(MIA_CHECKED, MIA_CHECKED));
               idCheckedItem = SHORT1FROMMP(mp1);
```

```
            switch SHORT1FROMMP(mp1)  /* change sound type */
               {
               case ID_NOTESEL: fsSound = WA_NOTE; break;
               case ID_WARNINGSEL: fsSound = WA_WARNING; break;
               case ID_ERRORSEL: fsSound = WA_ERROR; break;
               }
            break;

         case ID_EXIT:               /* Exit selected */
            WinPostMsg(hwnd, WM_QUIT, NULL, NULL);
            break;
         }
      break;

   case WM_CREATE:
                                     /* get menu window handle */
      hwndMenu = WinWindowFromID(WinQueryWindow(hwnd, QW_PARENT,
                                     FALSE), FID_MENU);
      break;

   case WM_ERASEBACKGROUND:
      return TRUE;                   /* erase background */
      break;

   default:
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }

   return NULL;                      /* NULL / FALSE */
   }
```

```
/* ---------- */
/* BEEPSACC.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_BEEP_SUBMENU 100
#define ID_BEEP 101
#define ID_NOTESEL 103
#define ID_WARNINGSEL 104
#define ID_ERRORSEL 105
#define ID_EXIT 110
```

```
# ------------------
# BEEPSACC Make file
# ------------------

beepsacc.obj: beepsacc.c beepsacc.h
   cl -c -G2s -W3 -Zp beepsacc.c

beepsacc.res: beepsacc.rc beepsacc.h
   rc -r beepsacc.rc
```

```
beepsacc.exe: beepsacc.obj beepsacc.def
   link /NOD beepsacc,,NUL,os2 slibce,beepsacc
   rc beepsacc.res beepsacc.exe

beepsacc.exe: beepsacc.res
   rc beepsacc.res
```

```
; ------------
; BEEPSACC.DEF
; ------------

NAME         BEEPSACC   WINDOWAPI

DESCRIPTION 'Different beeps using a menu and keyboard accelerator'

PROTMODE
STACKSIZE   4096
```

```
/* ----------- */
/* BEEPSACC.RC */
/* ----------- */

#include <os2.h>
#include "beepsacc.h"

MENU ID_FRAMERC
   BEGIN
      SUBMENU "~Beep", ID_BEEP_SUBMENU
         BEGIN
            MENUITEM "Sound ~beep\tB", ID_BEEP
            MENUITEM SEPARATOR
            MENUITEM "~Note\tAlt+N", ID_NOTESEL, , MIA_CHECKED
            MENUITEM "~Warning\tAlt+W", ID_WARNINGSEL
            MENUITEM "~Error\tAlt+E", ID_ERRORSEL
         END

      MENUITEM "E~xit", ID_EXIT
      MENUITEM "~Help", 0, MIS_HELP
   END

ACCELTABLE ID_FRAMERC
   BEGIN
      "b", ID_BEEP
      "B", ID_BEEP
      "n", ID_NOTESEL, ALT
      "N", ID_NOTESEL, ALT
      "w", ID_WARNINGSEL, ALT
      "W", ID_WARNINGSEL, ALT
      "e", ID_ERRORSEL, ALT
      "E", ID_ERRORSEL, ALT
      VK_F10, ID_EXIT, VIRTUALKEY, ALT, CONTROL
   END
```

You'll see the accelerator table included in the BEEPSACC.RC resource file. Surprisingly, only one other modification is necessary to enable your program to respond to accelerator keys: the FCF_ACCELTABLE constant must be used in the *flCreateFlags* argument to WinCreateStdWindow. No new message processing need be installed in the program, since the same message is generated whether the user selects an item from a menu or invokes it by pressing the accelerator keys. The programmer gets a lot of benefit from accelerators, with very little extra work.

The Help Item

The BEEPSACC program incorporates another refinement: the use of a *help item*. This appears as the item "Help", which is the rightmost item on the action bar. It's placed there by the statement

```
MENUITEM "Help", 0, MIS_HELP
```

in the resource file.

There are several unusual aspects to this statement. First, the item ID is, by convention, set to 0. Second, the MIS_HELP identifier causes a WM_HELP message to be sent, instead of the usual WM_COMMAND.

Processing WM_HELP

When an application receives a WM_HELP message, as a result of the user selecting the "Help" item from the action bar, the usual response is to display appropriate help text on the screen. This text often appears in a separate window, which can be removed from the screen by clicking on a button, and which often permits the display of additional text if requested by the user. The PM's Help Manager facilities such as the WinAssociateHelpInstance and WinCreateHelpTable can be used for this purpose, by using APIs. For simplicity, the BEEPSACC example uses WinDrawText to write the help phrase.

The System Accelerator Table

The accelerator key for the "Help" menu item is F1. This is the key mandated by the SAA Common User Interface for help. Pressing it will generate

the WM_HELP message and bring up the help phrase on the screen. However, if you look at the accelerator table in BEEPSACC's resource file, you won't see any mention of the F1 key. Where is it defined as an accelerator?

It turns out that, in addition to the accelerator tables for each application, PM maintains its own internal *system accelerator table.* The F1 key is defined here, as are the accelerator keys in the System Menu (ALT-F5 is "Restore", ALT-F7 is "Move", and so on). Using this built-in table, you don't need to know anything about accelerators to enjoy the benefits of processing help messages generated by the F1 key.

GETTING FANCY WITH MENUS

We've covered the bread-and-butter aspects of menus; now let's examine some less common menu operations. The next example will show how to change the text of a menu item and how to enable and disable menu items dynamically. It will also show how to implement sub-submenus, and it will introduce a handy system service: timers.

Rotating Colors Example

The example program divides the screen into four quadrants, and colors each one differently, as shown in Figure 8-7. Then it rotates the colors, creating a pinwheel effect. The effect is particularly attractive when the program is minimized: the minimized window continues to display the rotating colors.

When the program is first loaded, there are three items on the main menu: "Start", "Options", and "Exit". "Exit" is the same as in previous examples: selecting it terminates the program.

The "Options" item produces a submenu with two items, "Colors" and "Delay". However, these items look a little unusual: each has an arrow pointing to the right. Selecting an item, "Colors", for example, generates another menu to the right of the item, as shown in Figure 8-8. This is a sub-submenu.

The "Colors" sub-submenu provides six possible ways to arrange the colors in the four quadrants. The first (reading clockwise) is red, red, red, white; the second is blue, blue, blue, red, and so on. The "Delay" submenu provides five different delays. These are used to initialize a system timer that controls the speed of rotation of the colors.

Selecting "Start" from the main menu has several effects. First, it changes the word "Start" to "Stop". Second, it *disables* the "Options" menu

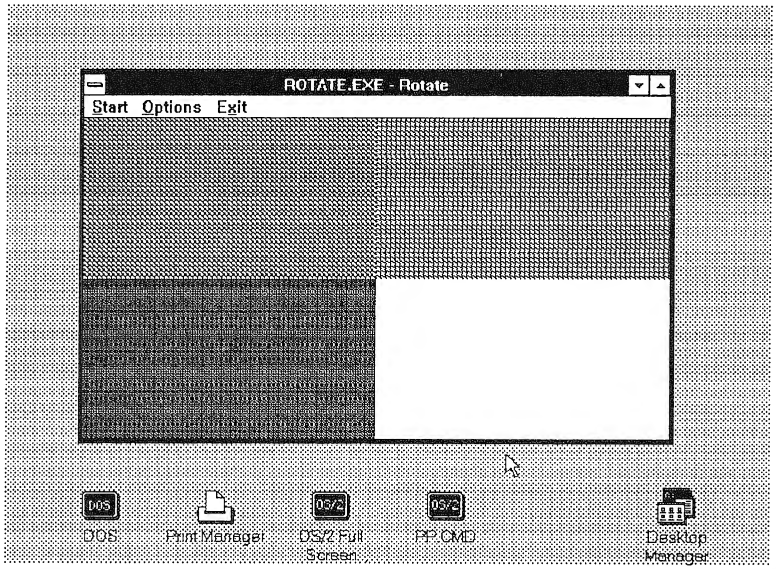


Figure 8-7. Output of the ROTATE program

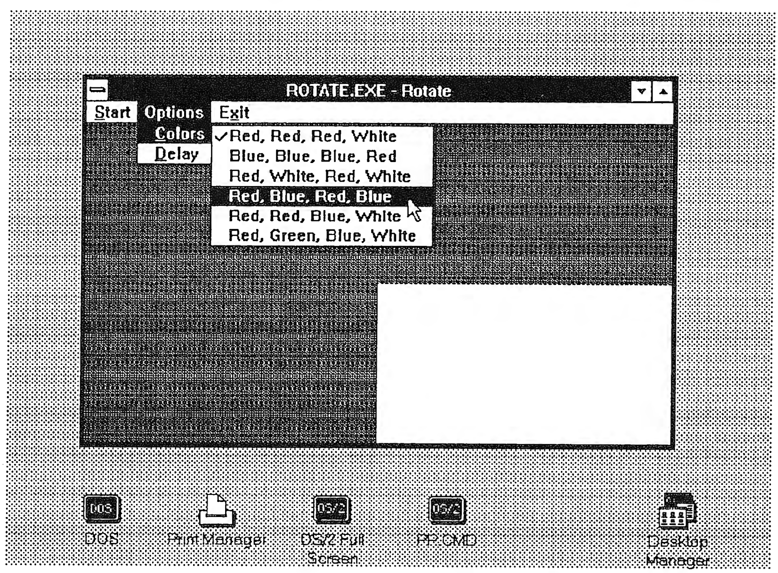


Figure 8-8. Sub-submenus

item. This means that the item appears in half-tone text and that it no longer sends messages when selected. The assumption is that you don't want to change any of the options while the colors are rotating. Finally, WinStartTimer is executed to start the colors rotating.

If you click on "Stop", the rotation stops, the word "Stop" changes back to "Start", and "Options" is enabled, becoming black again.

Overall Design of ROTATE

Here are the listings for ROTATE.C, ROTATE.H, ROTATE, ROTATE.DEF, and ROTATE.RC.

```
/* ------------------------------------------------- */
/* ROTATE - Using a menu to rotate color rectangles */
/* ------------------------------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "rotate.h"

HAB hab;                              /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                           /* handle for message queue */
   QMSG qmsg;                         /* message queue element */
   HWND hwnd, hwndClient;             /* handles for windows */

                                      /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_MENU;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);         /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);   /* create message queue */

                                      /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                      /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Rotate", 0L, NULL,
             ID_FRAMERC, &hwndClient);

                                      /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);            /* destroy frame window */
   WinDestroyMsgQueue(hmq);           /* destroy message queue */
   WinTerminate(hab);                 /* terminate PM usage */
```

```
   return 0;
   }

                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   static COLOR aclr[6][4] = {{CLR_RED, CLR_RED, CLR_RED, CLR_WHITE},
                              {CLR_BLUE, CLR_BLUE, CLR_BLUE, CLR_RED},
                              {CLR_RED, CLR_WHITE, CLR_RED, CLR_WHITE},
                              {CLR_RED, CLR_BLUE, CLR_RED, CLR_BLUE},
                              {CLR_RED, CLR_RED, CLR_BLUE, CLR_WHITE},
                              {CLR_RED, CLR_GREEN, CLR_BLUE, CLR_WHITE}};

   static HWND hwndMenu;
   static BOOL fRunning = FALSE;
   static SHORT iColorSet = 5;          /* current color set */
   static USHORT dtDelay = 250;         /* current time delay */
   LONG clrTemp;
   HPS hps;
   RECTL rcl, rclWindow;

   switch (msg)
      {
      case WM_COMMAND:
         switch SHORT1FROMMP(mp1)
            {
            case ID_START_STOP:
               if (!fRunning)
                  {                     /* "Start" */
                                        /* change "Start to Stop" */
                  WinSendMsg(hwndMenu, MM_SETITEMTEXT,
                     MPFROMSHORT(ID_START_STOP), "~Stop");
                                        /* disable options submenu */
                  WinSendMsg(hwndMenu, MM_SETITEMATTR,
                     MPFROM2SHORT(ID_OPTIONS_SUBMENU, FALSE),
                     MPFROM2SHORT(MIA_DISABLED, MIA_DISABLED));
                                        /* start timer */
                  WinStartTimer(hab, hwnd, TID_ROTATE, dtDelay);
                  }
               else
                  {                     /* "Stop" */
                                        /* stop timer */
                  WinStopTimer(hab, hwnd, TID_ROTATE);
                                        /* change "Stop" to "Start" */
                  WinSendMsg(hwndMenu, MM_SETITEMTEXT,
                     MPFROMSHORT(ID_START_STOP), "~Start");
                                        /* enable options submenu */
                  WinSendMsg(hwndMenu, MM_SETITEMATTR,
                     MPFROM2SHORT(ID_OPTIONS_SUBMENU, FALSE),
                     MPFROM2SHORT(MIA_DISABLED, ~MIA_DISABLED));
                  }
               fRunning = !fRunning;
               break;

            case ID_CSET0:
            case ID_CSET1:
            case ID_CSET2:
            case ID_CSET3:
            case ID_CSET4:
            case ID_CSET5:
```

```
                                    /* color set changed */
                                    /* un-check old color set */
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
               MPFROM2SHORT(iColorSet+ID_CSET0, TRUE),
               MPFROM2SHORT(MIA_CHECKED, ~MIA_CHECKED));
                                    /* check new color set */
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
               MPFROM2SHORT(SHORT1FROMMP(mp1), TRUE),
               MPFROM2SHORT(MIA_CHECKED, MIA_CHECKED));
            iColorSet = SHORT1FROMMP(mp1) - ID_CSET0;
            WinInvalidateRect(hwnd, NULL, FALSE);
            break;

         case ID_DELAY0000:
         case ID_DELAY0100:
         case ID_DELAY0250:
         case ID_DELAY0500:
         case ID_DELAY1000:
                                    /* delay changed */
                                    /* un-check old delay */
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
               MPFROM2SHORT(dtDelay + ID_DELAY0000, TRUE),
               MPFROM2SHORT(MIA_CHECKED, ~MIA_CHECKED));
                                    /* check new delay */
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
               MPFROM2SHORT(SHORT1FROMMP(mp1), TRUE),
               MPFROM2SHORT(MIA_CHECKED, MIA_CHECKED));
            dtDelay = (SHORT1FROMMP(mp1) - ID_DELAY0000);
            break;

         case ID_EXIT:
            WinPostMsg(hwnd, WM_QUIT, NULL, NULL);
            break;
         }
      break;

   case WM_TIMER:
      if (SHORT1FROMMP(mp1) == TID_ROTATE)
         {                          /* rotate colors */
         clrTemp = aclr[iColorSet][3];
         aclr[iColorSet][3] = aclr[iColorSet][2];
         aclr[iColorSet][2] = aclr[iColorSet][1];
         aclr[iColorSet][1] = aclr[iColorSet][0];
         aclr[iColorSet][0] = clrTemp;
         WinInvalidateRect(hwnd, NULL, FALSE);
         }
      break;

   case WM_PAINT:
      hps = WinBeginPaint(hwnd, NULL, NULL);
      WinQueryWindowRect (hwnd, &rclWindow);

                                    /* lower left */
      rcl.xLeft = rcl.yBottom = 0;
      rcl.xRight = rclWindow.xRight / 2;
      rcl.yTop = rclWindow.yTop / 2;
      WinFillRect(hps, &rcl,aclr[iColorSet][0]);

                                    /* upper left */
      rcl.yBottom = rcl.yTop;
      rcl.yTop = rclWindow.yTop;
      WinFillRect(hps, &rcl, aclr[iColorSet][1]);
```

```
                                         /* upper right */
         rcl.xLeft = rcl.xRight;
         rcl.xRight = rclWindow.xRight;
         WinFillRect(hps, &rcl, aclr[iColorSet][2]);

                                         /* lower right */
         rcl.yTop = rcl.yBottom;
         rcl.yBottom = 0;
         WinFillRect(hps, &rcl, aclr[iColorSet][3]);

         WinEndPaint(hps);
         break;

      case WM_CREATE:
                                         /* get menu window handle */
         hwndMenu = WinWindowFromID(WinQueryWindow(hwnd, QW_PARENT,
                                         FALSE), FID_MENU);
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                          /* NULL / FALSE */
   }
```

```
/* -------- */
/* ROTATE.H */
/* -------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1

#define ID_START_STOP 10

#define ID_OPTIONS_SUBMENU 11

#define ID_COLORS_SUBMENU 111
#define ID_CSET0 1111
#define ID_CSET1 1112
#define ID_CSET2 1113
#define ID_CSET3 1114
#define ID_CSET4 1115
#define ID_CSET5 1116

#define ID_DELAY_SUBMENU  20
#define ID_DELAY0000 20000
#define ID_DELAY0100 20100
#define ID_DELAY0250 20250
#define ID_DELAY0500 20500
#define ID_DELAY1000 21000

#define ID_EXIT 12

#define TID_ROTATE 1
```

```
# ----------------
# ROTATE make file
# ----------------

rotate.obj: rotate.c rotate.h
   cl -c -G2s -W3 -Zp rotate.c

rotate.res: rotate.rc rotate.h
   rc -r rotate.rc

rotate.exe: rotate.obj rotate.def
   link /NOD rotate,,NUL,os2 slibce,rotate
   rc rotate.res rotate.exe

rotate.exe: rotate.res
   rc rotate.res
```

```
; ----------
; ROTATE.DEF
; ----------

NAME        ROTATE WINDOWAPI

DESCRIPTION 'Rotate color rectangles'

PROTMODE
STACKSIZE   4096
```

```
/* --------- */
/* ROTATE.RC */
/* --------- */

#include <os2.h>
#include "rotate.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "~Start", ID_START_STOP
      SUBMENU "~Options", ID_OPTIONS_SUBMENU
         BEGIN
            SUBMENU "~Colors", ID_COLORS_SUBMENU
               BEGIN
                  MENUITEM "Red, Red, Red, White", ID_CSET0
                  MENUITEM "Blue, Blue, Blue, Red", ID_CSET1
                  MENUITEM "Red, White, Red, White", ID_CSET2
                  MENUITEM "Red, Blue, Red, Blue", ID_CSET3
                  MENUITEM "Red, Red, Blue, White", ID_CSET4
                  MENUITEM "Red, Green, Blue, White",ID_CSET5, ,MIA_CHECKED
               END
            SUBMENU "~Delay", ID_DELAY_SUBMENU
               BEGIN
                  MENUITEM "Full Speed", ID_DELAY0000
                  MENUITEM "0.1 Second delay", ID_DELAY0100
                  MENUITEM "0.25 Second delay", ID_DELAY0250, , MIA_CHECKED
                  MENUITEM "0.5 Second delay", ID_DELAY0500
                  MENUITEM "1 Second delay", ID_DELAY1000
               END
```

```
      END
   MENUITEM "E~xit", ID_EXIT
END
```

With so many things going on, the .C listing for this program is longer than most. However, if you follow the action taken in response to particular messages, it's not hard to figure out. We'll look at the program's overall response to messages first, and then examine the various new topics in more detail.

We'll start at the bottom of the listing and work up.

Processing WM_CREATE

We need the handle of the menu window, so we obtain it when the program receives a WM_CREATE message. This is done using WinQueryWindow and WinWindowFromID, as described in earlier examples.

Processing WM_PAINT

Repainting the window involves finding out how big the client window is, dividing this area into four quadrants, and filling each one with a different color.

The colors are stored in a two-dimensional array called *aclr,* which mirrors the choices available in the "Colors" menu. The first index variable, *iColorSet,* points to which set of four colors has been selected by the user. The second index varies from 0 to 3 to specify the particular quadrant the color will be placed in.

A new function, WinFillRect, is used to color each quadrant with the appropriate color.

Fill Rectangle

```
BOOL WinFillRect(hps, pcrl, clr)
HPS hps          Presentation space containing rectangle
PRECTL pcrl      Rectangle to be filled
COLOR clr        Color (CLR_RED, etc.) or index to color table

Returns: TRUE if successful, FALSE if error
```

The color provided to this function is usually represented by constants like CLR_RED, CLR_BLUE, and so on. In ROTATE these values are taken from the *aclr* array.

Processing WM_TIMER

This message is received whenever a timer delay has expired, which is determined by the value in the variable *dtDelay.* Each time the WM_TIMER is received, the colors in the *aclr* array, for a particular value of *iColorSet,* are rotated. WinInvalidateRect is then used to cause PM to generate a WM_PAINT message so the window will be redrawn with the new colors.

Processing WM_COMMAND

The bulk of the message processing in this program is the result of WM_COMMAND messages received as the result of menu selections.

An item ID of ID_START_STOP is received when either the "Start" or "Stop" item is selected. (This is really the same item, with the text changed.) A flag, *fRunning,* keeps track of whether the colors are rotating or not. Switching between the running and not running states involves three actions: changing the text of "Start" to "Stop" or vice versa, enabling and disabling "Options", and starting and stopping the timer.

Item IDs of ID_CSET0 through ID_CSET5 are received when the user selects a new color set from the "Colors" menu. These IDs are numbered contiguously, which simplifies setting *iColorSet* to the new value. The old color set must also be unchecked, and the new one checked, in the "Colors" menu.

Item IDs of ID_DELAY0000 through ID_DELAY1000 are received when the user selects a new value from the "Delay" menu. These IDs have the same value as the delay they represent (plus a constant), which simplifies setting the variable *dtDelay* to the appropriate value. The old delay must be unchecked and the new one checked in the "Delay" menu.

We've seen in previous examples how to process ID_EXIT, generated when the user selects "Exit".

Now that we've explored the program in general, let's see how the various new features—changing menu item text, enabling and disabling items, sub-submenus, and system timers—are implemented.

Changing Menu Item Text

We saw in the BEEPS example how to check and uncheck menu items. Changing the text of an item is similar. It involves sending an MM_SETITEMTEXT message with WinSendMsg. The first half of the *mp1*

argument of this message is the menu item ID, and the entire *mp2* argument is a pointer to the new text to be placed in the item.

ROTATE makes text changes when the ID_START_STOP is received in a WM_COMMAND message. "Start" is changed to "Stop" or vice versa, depending on the state of the *fRunning* flag.

Disabling and Enabling Menu Items

At the same time "Start" is changed to "Stop", the "Options" item on the main menu is being disabled. When it's disabled, its text is shaded half-tone (gray), and it stops generating messages, no matter how insistently the user clicks on it. (Submenus can be called up by the user from a disabled item, but attempting to select items from them elicits only a beep.)

Disabling and enabling a menu item is again similar to checking it and unchecking it. The MM_SETITEMATTR message is sent with WinSendMsg. The arguments to this function are exactly the same as those used for checking and unchecking, except that MIA_DISABLED is used instead of MIA_CHECKED.

Sub-Submenus

Sub-submenus are implemented in the resource file, as a natural extension of the way submenus are implemented. In the place where you would ordinarily insert a single MENUITEM statement in a submenu, you insert instead a SUBMENU keyword followed by a list of MENUITEM statements delimited by BEGIN and END statements. In other words, you plug in a complete submenu instead of an item. You can see how this looks in ROTATE.RC. Items on the sub-submenus send messages with their ID number when they are selected, just as items on submenus and on the top-level menu do.

Timers

The Presentation Manager makes available a number of *timers* that an application can use. After it is started, a timer generates WM_TIMER messages at fixed intervals until it is stopped. The interval is specified in milliseconds. Its minimum length is determined by the system clock; on most systems it's about 55 milliseconds. (These clock intervals are often called "timer ticks.") The maximum interval length is limited by the variable used to hold it—currently this is a USHORT, so the maximum is 65,536 milliseconds, which is one minute and 5 seconds.

An application can use the WM_TIMER messages in a variety of ways. In the present example we use them to time the rotation of the color quadrants. They can also be used to create clocks and similar time-dependent devices.

However, don't count on the timer messages arriving at precisely spaced intervals. First, the messages are only as precise as the system clock, so that any interval you specify will be rounded to a multiple of 55 milliseconds. Also, the timer messages can be delayed if PM is busy with something else. If you're implementing a clock, use timer messages as a *suggestion,* rather than the fact, that (for example) a second has passed. Then check the system clock by looking in the global information segment with the kernel call DosGetInfoSeg to see how much time has *really* passed.

Another problem is that the system has a limited number of timers available. If other applications are using them, you may not be able to get one when you need it. Your application should be prepared to deal with this situation.

If a timer message arrives while others with the same ID are still waiting in the message queue, it will be combined with them. Thus there is at most one timer message in the queue at any time. This message may indicate that several messages were sent, but you were too busy to process them.

Starting the Timer

A timer is started with the WinStartTimer function.

Start a Timer

```
USHORT WinStartTimer(hab, hwnd, idTimer, dtTimeout)
HAB hab             Anchor block handle
HWND hwnd           Handle of window requesting timer
USHORT idTimer      Timer ID
USHORT dtTimeout    Interval in milliseconds (0=minimum)

Returns: TRUE if successful, FALSE if error
```

An error return may indicate no timer was available.

The ROTATE example starts the timer in response to the user selecting "Start" from the menu.

The WM_TIMER Message

While it is running, the timer generates WM_TIMER messages at the interval specified in WinStartTimer. The ROTATE example responds to these messages by rotating the color quadrant colors.

Be careful handling WM_TIMER messages. Other timers—started by the system and other applications—may also be generating WM_TIMER messages. You should therefore check the ID of the timer to be sure it is yours, before you process the message.

Stopping the Timer

The timer is stopped with the WinStopTimer function.

Stop a Timer

```
BOOL WinStopTimer(hab, hwnd, idTimer)
HAB hab             Anchor block handle
HWND hwnd           Window handle
USHORT idTimer      Timer ID

Returns: TRUE if successful, FALSE if error
```

ROTATE stops the timer when "Stop" is selected from the menu. An application should always stop a timer when it is no longer needed, so that the resources it uses can be returned to the system. Timers tend to slow down the system, since every timer is updated at every clock tick, so minimizing their use will make the user happier.

OTHER MENU AND ACCELERATOR OPERATIONS

Other more complex manipulations are possible with menus and accelerators.

An application can insert a completely new item into an existing menu. This is done by filling in a MENUITEM structure with the position, style, and other features of the item, and then sending an MM_INSERTITEM message to the menu window. Existing items can be deleted by sending an MM_DELETEITEM message. This sort of manipulation is useful when the application doesn't know in advance what to place on the menu. For example, a word processing program might need to find out from the system what fonts were available, before displaying the choices on a menu.

Entire menus can be loaded from a resource while the program is running, using the WinLoadMenu function. This is analogous to using WinLoadPointer to load a pointer dynamically. Menus can also be created dynamically, without using a resource, using WinCreateMenu.

Accelerator tables, both the system accelerator table and those belonging to applications, can also be changed dynamically, using APIs like WinLoad-AccelTable and WinSetAccelTable. We won't pursue these ideas further here.

9

DIALOG BOXES

If menus are the first form of interaction a user encounters in a PM application, dialog boxes are the second. Selecting an item from a menu frequently calls up a dialog box. The dialog box then elicits more detailed information about the user's intentions.

In this chapter we'll first examine a simplified version of a dialog box called a message box. Next we'll look at a simple dialog box, and then at a more complex dialog box. Finally we'll look at a somewhat unusual use for a dialog box in a game program.

THE USER'S VIEW OF DIALOG BOXES

To the user, a dialog box is a group of controls—such as static text, buttons, and entry fields—surrounded by a border. The dialog border usually consists of a colored line surrounded by a thin black line.

A dialog box may appear in response to an action taken by the user. For instance, the user might select a "Save as . . ." item from a "File" menu. This would invoke a dialog box that would contain an entry field into which the user could type the name of the file to be saved and perhaps set other parameters. The ellipsis (. . .) following a menu item is used, by convention, to indicate that selecting the item will invoke a dialog box.

Dialog boxes also appear in response to internal states reached by the application. If the application determines that a disk drive is not ready, for example, it might display a dialog box with the text "Drive not ready" and push buttons to select "OK" or "Cancel".

A dialog box (but not a message box) can incorporate all the kinds of controls described in Chapter 6, including static windows containing text or graphics, push buttons, radio buttons, check boxes, entry fields, MLEs, combo boxes, and scroll bars. (They may also contain custom window classes defined by the program.) Dialog boxes can be very simple, with only a few controls, or they can be quite complex.

MESSAGE BOXES

A message box is a special, limited form of dialog box. It is easier to program than a real dialog box, but it is limited in the kinds of controls it can use. Specifically, it can contain one text phrase, up to three push buttons, and optionally an icon, but nothing else. Even with these limitations, it is surprisingly useful.

We'll demonstrate message boxes using a modified version of the BEEP program from the previous chapter. The new example, MSGBOX, places two menu items in the action bar, just as BEEP did: "Beep" and "Exit". As before, clicking on "Beep" causes a tone to sound. However, clicking on "Exit" brings up a message box with the text "Are you sure?" and two push buttons: "Yes" and "No", as shown in Figure 9-1. Clicking on "Yes" terminates the program, while clicking on "No" simply removes the dialog box from the application's window.

MSGBOX also features a "Help" item on the action bar. Clicking on this item invokes another message box. This one, which says "This is the help message," has a single push button reading "Cancel". Clicking on this button makes the box go away. Such a box might convey real help information in a simple application. (In a more complex application you would probably put the help information in a separate window.)

Modal and Modeless

Notice that while a message box is on the screen, you can't interact with the other features of the application. Clicking on menu items or on anything else in the application's window results in an annoying beep. You must make the message box disappear before you can carry out any other

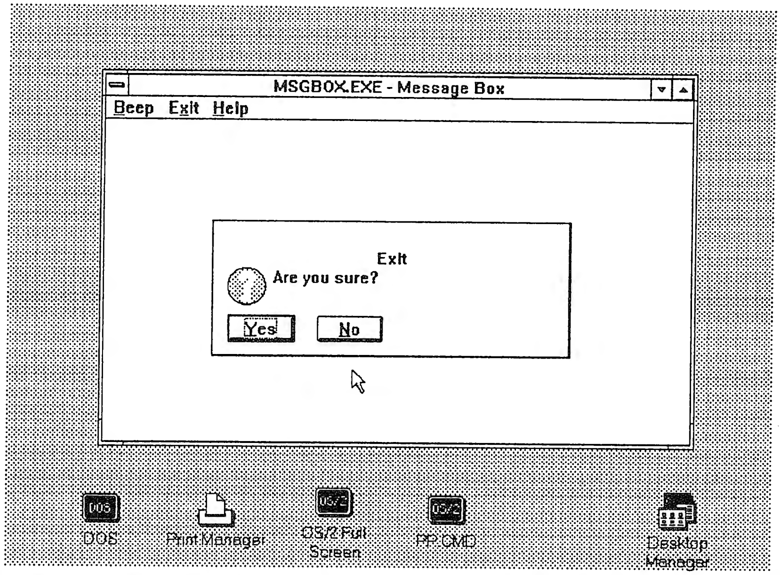


Figure 9-1. Output of MSGBOX program

operations with your program. The message box is *modal:* While it is on the screen, the application is in a different mode of operation. In this mode, only the message box can receive user input; other windows of the application cannot.

When to Use Modal Dialogs

There's a certain philosophy underlying the use of modal features in PM. Generally you want an application to work the same way at all times. Dragging the title bar should always move the window, clicking on the "Save as" menu item should always evoke the same dialog box, and so on. If the same action can have different effects at different times, there is the possibility the user will become confused and frustrated ("Why can't I bring up the &!@X%!& Help menu?"). So modal operation is generally not desirable.

However, in some cases modal operation is acceptable and even necessary. If the application is about to run out of memory, for example, the user should be informed of this fact immediately, and other operations should be suspended until the problem is acknowledged. Modal operation is satisfactory if it is obvious to the user when a different mode is in effect.

Message boxes and dialog boxes signal the new mode by their very presence on the screen. So long as they are large enough to be obvious and are placed at the top of the Z-axis, the user should not become confused.

Application and System Modal

There are several kinds of modal operation. A message or dialog box can prevent the rest of its own application from receiving user input messages, as MSGBOX does. This is called *application modal.* It can also prevent *all* the applications in the system from receiving user input messages. This is called *system modal,* and is useful when an application is reporting a systemwide catastrophe, such as, "Your hard disk is full" or "Your printer is on fire." We won't be concerned with system modal dialog boxes.

A *modeless* dialog box can also be created. It can be displayed without preventing other windows from receiving user input messages. The last example in this chapter demonstrates a modeless dialog box.

Using the Keyboard with a Message Box

As with other PM features, the user can control a message box entirely from the keyboard. Once a message box appears, you can move the keyboard focus from one button to another by pressing the TAB key or the arrow keys. You can tell which button has the focus by the dotted outline around the text. Pressing ENTER selects the current default button (the one with the black border). Pressing ESCAPE is equivalent to pressing a "Cancel" button if there is one.

Programming the Message Box

Including a message box in your application involves nothing more than a call to a new function, so most of the next application will be familiar. Here are the listings for MSGBOX.C, MSGBOX.H, MSGBOX, MSGBOX.DEF, and MSGBOX.RC.

```
/* ---------------------------- */
/* MSGBOX - Using a message box */
/* ---------------------------- */

#define INCL_WIN
#include <os2.h>

#include "msgbox.h"
```

```
static HWND hwndFrame;

USHORT cdecl main(void)
   {
   HAB hab;                           /* handle for anchor block */
   HMQ hmq;                           /* handle for message queue */
   QMSG qmsg;                         /* message queue element */
   HWND hwndClient;                   /* handles for windows */

                                      /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_MENU;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);         /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);   /* create message queue */

                                      /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, 0L, 0);

                                      /* create standard window */
   hwndFrame = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
                  szClientClass, " - Message Box", 0L, NULL,
                  ID_FRAMERC, &hwndClient);

                                      /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwndFrame);       /* destroy frame window */
   WinDestroyMsgQueue(hmq);           /* destroy message queue */
   WinTerminate(hab);                 /* terminate PM usage */

   return 0;
   }

                                      /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   switch (msg)
      {
      case WM_HELP:
         WinMessageBox(
            HWND_DESKTOP,             /* parent */
            hwndFrame,                /* owner (frame window) */
            "This is the help message.",   /* text */
            "Help",                   /* title */
            0,                        /* ID */
            MB_CANCEL);               /* cancel button */
         break;

      case WM_COMMAND:
         switch (COMMANDMSG(&msg)->cmd)
            {
            case ID_BEEP:             /* Beep selected */
```

```
            WinAlarm(HWND_DESKTOP, WA_NOTE);
            break;

         case ID_EXIT:                  /* Exit selected */
            if (WinMessageBox(
                  HWND_DESKTOP,         /* parent */
                  hwndFrame,            /* owner (frame window) */
                  "Are you sure?",      /* text */
                  "Exit",               /* title */
                  0,                    /* ID */
                  MB_YESNO |            /* yes & no buttons */
                  MB_ICONQUESTION)      /* question icon */
                == MBID_YES)
               WinPostMsg(hwnd, WM_QUIT, NULL, NULL);
            break;
         }
      break;

   case WM_ERASEBACKGROUND:
      return TRUE;                      /* erase background */
      break;

   default:
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }

return NULL;                            /* NULL / FALSE */
}
```

```
/* -------- */
/* MSGBOX.H */
/* -------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_BEEP 101
#define ID_EXIT 102
```

```
# ----------------
# MSGBOX Make file
# ----------------

msgbox.obj: msgbox.c msgbox.h
   cl -c -G2s -W3 -Zp msgbox.c

msgbox.res: msgbox.rc msgbox.h
   rc -r msgbox.rc

msgbox.exe: msgbox.obj msgbox.def
   link /NOD msgbox,,NUL,os2 slibce,msgbox
   rc msgbox.res msgbox.exe

msgbox.exe: msgbox.res
   rc msgbox.res
```

```
; ----------
; MSGBOX.DEF
; ----------

NAME        MSGBOX  WINDOWAPI

DESCRIPTION 'Using a message box'

PROTMODE
STACKSIZE   4096
```

```
/* --------- */
/* MSGBOX.RC */
/* --------- */

#include [os2.h]
#include "msgbox.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "~Beep", ID_BEEP
      MENUITEM "E~xit", ID_EXIT
      MENUITEM "~Help", 0, MIS_HELP
   END
```

The *main* function in MSGBOX.C and the other files should offer no surprises. The new features are in the client window procedure.

The WinMessageBox Function

The WinMessageBox function is invoked twice. The first time is the result of a WM_HELP message, which is received when the user selects the "Help" item. The second time is as a result of a WM_COMMAND message with an ID of ID_EXIT, received when the user selects "Exit" from the action bar.

Create Message Box Window

```
USHORT WinMessageBox(hwndParent, hwndOwner, pszText, pszCaption,
                     idWindow, fsStyle)
HWND hwndParent   Parent of message box
HWND hwndOwner    Owner of message box
PSZ pszText       Message text
PSZ pszCaption    Title
USHORT idWindow   ID of message box
USHORT fsStyle    Style (MB_OK, MB_ICONHAND, etc.)

Returns: ID of button pressed (see table of MBID_ values)
```

The parent of the message box is usually the desktop window, HWND_DESKTOP. We want the complete message box to be visible, no matter what size the application's top-level window is. Using the desktop as the parent avoids the message box being clipped to the frame window's boundaries, as it would be if the frame window were its parent. PM automatically positions message boxes in the middle of the screen.

The owner of the message box as specified by the *hwndOwner* argument to WinMessageBox, and all its descendants, are disabled until WinMessageBox returns. To conform to the SAA Common User Access guidelines, you should specify the application's top-level frame window as the message box's owner. Doing this will disable all the application's windows. Thus in the example program we specify *hwndFrame* as the message box's owner.

String constants represent the *pszText* item in both calls to WinMessageBox: "This is the help message" and "Are you sure?" (in a full-scale application such strings would probably be stored as a string resource). The *pszTitle* argument supplies a title that appears at the top of the message box, above the message text.

The *idWindow* argument is an identifer to be used with hooks, a programming concept we ignore here.

Message Box Styles

A key argument to WinMessageBox is *fsStyle:* It determines how the message box will look. Constants for this argument are divided into three groups. These constants, one from each group, can be ORed together. First, a constant specifies the number and kinds of buttons:

Message Box Style	Buttons Generated
MB_OK	"OK" button
MB_OKCANCEL	"OK" and "Cancel" buttons
MB_RETRYCANCEL	"Retry" and "Cancel" buttons
MB_ABORTRETRYIGNORE	"Abort", "Retry", and "Ignore" buttons
MB_YESNO	"Yes" and "No" buttons
MB_YESNOCANCEL	"Yes", "No", and "Cancel" buttons
MB_ENTER	"Enter" button
MB_ENTERCANCEL	"Enter" and "Cancel" buttons

Next, an icon can be added to the message box, using one of these identifiers:

Message Box Style	Icon Generated
MB_ICONQUESTION	Question mark icon
MB_ICONEXCLAMATION	Exclamation point icon
MB_ICONASTERISK	Asterisk icon
MB_ICONHAND	Hand icon
MB_NOICON	No icon

The icon is always added in the same place in the message box.

Identifiers can also change the default push button. This button is drawn with a heavy black border and is given the keyboard focus when the box first appears. Normally the first button is the default, and the message box is application modal.

Message Box Style	Meaning
MB_DEFBUTTON1	First button is default (default)
MB_DEFBUTTON2	Second button is default
MB_DEFBUTTON3	Third button is default
MB_APPLMODAL	Application modal message box (default)
MB_SYSTEMMODAL	System modal message box
MB_HELP	Box contains "Help" button
MB_MOVEABLE	Box is moveable

Returning from WinMessageBox

The return value from WinMessageBox is determined by what button the user pressed to get rid of the box. In the case of the box invoked by WM_HELP in the MSGBOX example, there is only one button, so we don't worry about the return value. The box invoked by ID_EXIT, however, has two buttons, "Yes" and "No". If "Yes" is selected, we want to terminate the program, so we check if the return value is equal to a constant, MBID_YES. Here are the possible return values from WinMessageBox:

Return Value	Button Pressed by User
MBID_OK	"OK"
MBID_CANCEL	"Cancel" or ESCAPE key
MBID_ABORT	"Abort"
MBID_RETRY	"Retry"
MBID_IGNORE	"Ignore"
MBID_YES	"Yes"
MBID_NO	"No"
MBID_ENTER	"Enter"
MBID_ERROR	WinMessageBox failed

If the user clicked on "No" in the "Are you sure?" box, then we take no action.

What Makes It Modal?

Notice that once our application has called WinMessageBox, it must wait for the function to return, and the function won't return until the user has pressed one of the buttons in the message box. This is what makes message boxes modal: the application must wait for the user to move out of "message box mode."

The COMMANDMSG Macro

There's a new wrinkle in MSGBOX. In previous examples the *switch* statement that was invoked by the WM_COMMAND message in the client window procedure looked like this:

```
case WM_COMMAND:
   switch SHORT1FROMMP(mp1)
      {
      case ID_XXX:
      /* (other case statements) */
```

The first half of *mp1* carries the ID of the control that sent the WM_COMMAND message. This is sometimes called the *command value.* The *switch* statement causes different actions depending on this ID.

In MSGBOX the equivalent code looks like this:

```
case WM_COMMAND:
   switch (COMMANDMSG(&msg)->cmd)
      {
      case ID_XXX:
      /* (other case statements) */
```

The COMMANDMSG macro is defined in PMWIN.H. It makes use of the fact that the arguments to a window procedure are stored on the stack. It points to the appropriate parts of the parameters using names defined in the following structure, which is also defined in PMWIN.H:

```
struct _COMMANDMSG {
   USHORT source;     /* second half mp2 */
   BOOL fMouse;       /* first half mp2 */
   USHORT cmd;        /* second half mp1 */
   USHORT unused;     /* first half mp2 */
};
```

This structure is specific to the WM_COMMAND, WM_HELP, and WM_SYSCOMMAND messages. The parameters of other messages would be given different names.

Using this macro makes it possible to use more meaningful names for the parts of the message parameters *mp1* and *mp2*. Names like *COMMANDMSG(&msg)–>cmd* and *COMMANDMSG(&msg)–>source* are at least somewhat more easily recognized as the command value and source value than are *SHORT1FROMMP(mp1)* and *SHORT1FROMMP(mp2)*. We've avoided use of this macro until this point because of its nonintuitive appearance, but it's a common feature in PM programs, so we'll employ it in this and subsequent chapters.

Several other macros are used in similar ways to deal with different kinds of messages. These are CHARMSG, which accesses the WM_CHAR message; and MOUSEMSG, which accesses the WM_MOUSEMOVE and WM_BUTTON1DOWN family of messages.

SIMPLE DIALOG BOXES

Our next example places two items on the action bar: "Beep" and "Exit". (There is no "Help" item.) Clicking on "Beep" causes a beep. Clicking on

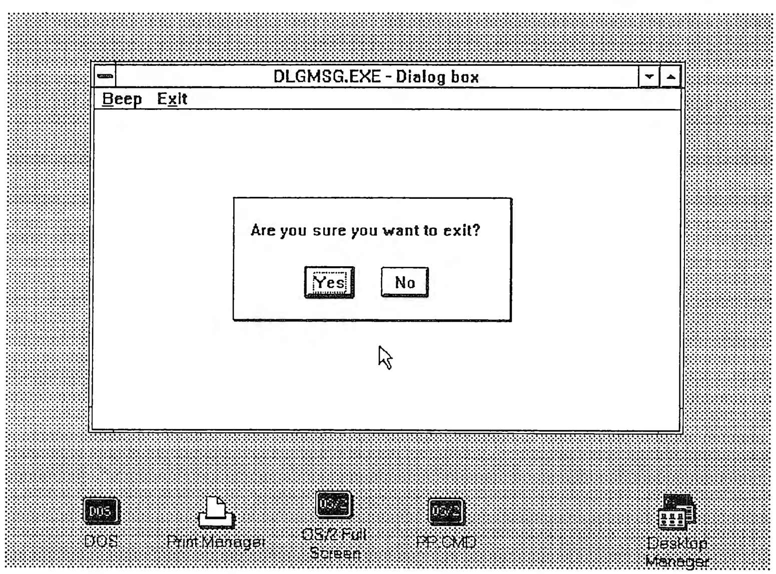


Figure 9-2. Output of the DLGMSG program

"Exit" brings up a dialog box with the text "Are you sure you want to exit?" and two push buttons, "Yes" and "No", as shown in Figure 9-2. This is a real dialog box, not a message box, although it looks similar to the message box in the last example (without the icon). The difference is in the programming.

Here are the files for this example: DLGMSG.C, DLGMSG.H, DLGMSG, DLGMSG.DEF, and DLGMSG.RC.

```
/* ---------------------------- */
/* DLGMSG - A simple dialog box */
/* ---------------------------- */

#define INCL_WIN
#include <os2.h>

#include "dlgmsg.h"

HWND hwndFrame;

USHORT cdecl main(void)
   {
   HAB hab;                            /* handle for anchor block */
   HMQ hmq;                            /* handle for message queue */
   QMSG qmsg;                          /* message queue element */
   HWND hwndClient;                    /* handles for windows */

                                       /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_MENU;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);          /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);    /* create message queue */

                                       /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, 0L, 0);

                                       /* create standard window */
   hwndFrame = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
                  szClientClass, " - Dialog box", 0L, NULL,
                  ID_FRAMERC, &hwndClient);

                                       /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwndFrame);        /* destroy frame window */
   WinDestroyMsgQueue(hmq);            /* destroy message queue */
   WinTerminate(hab);                  /* terminate PM usage */

   return 0;
   }

                                       /* client window procedure */
```

```
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   switch (msg)
      {
      case WM_COMMAND:
         switch (COMMANDMSG(&msg)->cmd)
            {
            case ID_BEEP:                /* Beep selected */
               WinAlarm(HWND_DESKTOP, WA_NOTE);
               break;

            case ID_EXIT:                /* Exit selected */
               if (WinDlgBox(
                      HWND_DESKTOP,      /* parent */
                      hwndFrame,         /* owner */
                      DlgProc,           /* dialog proc */
                      NULL,              /* dialog template in EXE */
                      ID_DLGBOX,         /* dialog template ID */
                      NULL)              /* no params */
                  == ID_YES)             /* if yes, quit */
                  WinPostMsg(hwnd, WM_QUIT, NULL, NULL);
               break;
            }
         break;

      case WM_ERASEBACKGROUND:
         return TRUE;                    /* erase background */
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                          /* NULL / FALSE */
   }

                                         /* dialog window procedure */
MRESULT EXPENTRY DlgProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   return WinDefDlgProc(hwnd, msg, mp1, mp2);
   }
```

```
/* -------- */
/* DLGMSG.H */
/* -------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);
MRESULT EXPENTRY DlgProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_BEEP 101
#define ID_EXIT 102

#define ID_DLGBOX 200
#define ID_MSG 201
```

```
#define ID_-YES 211
#define ID_NO 212
```

```
# ----------------
# DLGMSG Make file
# ----------------

dlgmsg.obj: dlgmsg.c dlgmsg.h
   cl -c -G2s -W3 -Zp dlgmsg.c

dlgmsg.res: dlgmsg.rc dlgmsg.h
   rc -r dlgmsg.rc

dlgmsg.exe: dlgmsg.obj dlgmsg.def
   link /NOD dlgmsg,,NUL,os2 slibce,dlgmsg
   rc dlgmsg.res dlgmsg.exe

dlgmsg.exe: dlgmsg.res
   rc dlgmsg.res
```

```
; ----------
; DLGMSG.DEF
; ----------

NAME        DLGMSG  WINDOWAPI

DESCRIPTION 'A simple dialog box'

PROTMODE
STACKSIZE   4096
```

```
/* --------- */
/* DLGMSG.RC */
/* --------- */

#include <os2.h>
#include "dlgmsg.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "~Beep", ID_BEEP
      MENUITEM "E~xit", ID_EXIT
   END

DLGTEMPLATE ID_DLGBOX
   BEGIN
      DIALOG "", ID_DLGBOX, 80, 50, 150, 50
         BEGIN
            LTEXT "Are you sure you want to exit?", ID_MSG, 10, 30, 130, 12
            DEFPUSHBUTTON "Yes", ID_YES, 40, 8, 28, 14
            PUSHBUTTON "No", ID_NO, 82, 8, 28, 14
         END
   END
```

The *main* function is identical to that in MSGBOX.C. The client window procedure, however, uses a new API: WinDlgBox. There is also—for the first time since Chapter 4—a new procedure: the *dialog window procedure.*

The WinDlgBox Function

The WM_COMMAND message with an ID value of ID_EXIT is handled differently in this example than it was in MSGBOX. In that example we invoked WinMessageBox, while in the current example we invoke WinDlgBox.

Load and Process Dialog Box

```
USHORT WinDlgBox (hwndParent, hwndOwner, pfnDlgProc, hmod, idDlg,
                  pCreateParams)
HWND hwndParent          Parent window of dialog box
HWND hwndOwner           Owner window of dialog box
PFNWP pfnDlgProc         Dialog window procedure
HMODULE hmod             Resource module (NULL = .EXE file)
USHORT idDlg             ID of dialog window in resource file
PVOID pCreateParams      Dialog box specific control data

Returns: Value supplied by WinDismissDlg, or DID_ERROR if error
```

This function creates a dialog box. A dialog "box," like many visual aspects of PM, is actually a window. We'll refer to dialog boxes as dialog windows when we want to emphasize the programmer's view of it as opposed to the user's view.

Parameters of WinDlgBox

The parent of the dialog window is HWND_DESKTOP, the desktop window. As in the previous example, we don't want the dialog box clipped at the boundaries of the frame window.

The dialog window's owner is the frame window, *hwndFrame.* This disables all windows in the application, as was done earlier in the MSGBOX example. The *pfnDlgProc* argument to WinDlgBox is the address of the dialog window procedure, which we'll look at in the next section. The *hmod* argument is the same as that we've seen in other resources. It specifies a DLL holding the resource; if it's NULL, the resource is in the application's .EXE file, as it is here.

The *idDlg* argument is used to identify the dialog template in the resource file (which we'll look at soon). The *pCreateParams* argument is used to pass application-specific data to the dialog procedure along with the WM_INITDLG message. (This is similar to the control data passed to a window procedure with the WM_CREATE message.) We don't use this in our example, so it's set to NULL.

Return Value of WinDlgBox

In our example the return value from WinDlgBox is set by the dialog window procedure. If it's ID_YES, the program terminates. We'll see in the next section where this return value originates.

The Dialog Window Procedure

A dialog window exists to process messages from its ownees, which are control windows—push buttons and so on—placed in the dialog box. The processing of these messages is carried out by a *dialog window procedure.* This procedure is similar to a client window procedure. It is a window procedure located in the application.

Like the client window procedure, the dialog window procedure is defined as MRESULT EXPENTRY.

Default Processing

In our first example of a client window procedure, in Chapter 5, we showed a procedure with only one line of code: the function WinDefWindowProc. This function handles all messages not explicitly handled by the client window procedure. Similarly, in DLGMSG the dialog window procedure is only one line long. However, the function that does default message processing for a dialog window is different: it's WinDefDlgProc. By calling this function we process any message received by the dialog window that we don't want to process ourselves.

Perform Default Processing of Dialog Window Messages

```
MRESULT WinDefDlgProc(hwndDlg, msg, mp1, mp2)
HWND hwndDlg      Dialog window handle
USHORT msg        Message identifier number
MPARAM mp1        Message parameter 1 (message specific)
MPARAM mp2        Message parameter 2 (message specific)

Returns: Message return value
```

The arguments to WinDefDlgProc are similar to those for WinDef-WindowProc.

Behind the Scenes

In DLGMSG we use this default processing to provide the return value to WinDlgBox. How does this work? WinDlgBox creates the dialog box and will return only when the dialog procedure issues a certain API: Win-DismissDlg. When calling WinDismissDlg, the dialog procedure specifies the value that WinDlgBox will return.

In our example, when one of the push buttons in the dialog box is pressed, the button control window sends a WM_COMMAND message to its owner, the dialog window. That is, it calls our dialog window procedure *DlgProc.* This procedure forwards the message to WinDefDlgProc, which in this example handles all the dialog window messages. For a WM_COMMAND message the default action performed internally by Win-DefDlgProc is to call WinDismissDlg, specifying the ID of the button that sent the WM_COMMAND as the value to be returned. Thus, WinDismiss-Dlg removes the dialog box and returns the ID of the control to Win-DlgBox. In this example WinDismissDlg is executed internally by WinDefDlgProc. In the next example we'll execute it ourselves, and this sequence may be easier to understand.

When WinDlgBox returns, the dialog window is automatically destroyed. The flow of messages and control in DLGBOX is shown in Figure 9-3.

Creating Dialog Boxes by Hand

How does WinDlgBox know what kind of dialog box to create? It uses a template in the resource file. This template specifies everything about the dialog box, including its size, the number and kinds of controls, and the position of the controls within the box.

There are several ways to create a dialog box resource. You can type it directly into the resource file, or you can use the dialog editor utility. (You can also have your program create a dialog box dynamically in memory, but we'll ignore this possibility here.) In this example we typed the dialog resource by hand, just as we did with the MENU specification and the other lines in the resource file. As you can see in the .RC file, the dialog resource looks like this:

```
DLGTEMPLATE ID_DLGBOX
 BEGIN
```

```
DIALOG "", ID_DLGBOX, 80, 50, 150, 50
 BEGIN
  LTEXT "Are you sure you want to exit?", ID_MSG, 10, 30, 130, 12
  DEFPUSHBUTTON "Yes", ID_YES, 40, 8, 28, 14
  PUSHBUTTON "No", ID_NO, 82, 8, 28, 14
 END
END
```

The DLGTEMPLATE Keyword

The DLGTEMPLATE keyword specifies a dialog *template.* The template is the data stored in the resource. This data consists of a header, which is a structure of type DLGTEMPLATE, followed by an array of dialog window items. These are a frame window and the various control windows that are its ownees. You don't usually need to be concerned with the format of the data in the header.

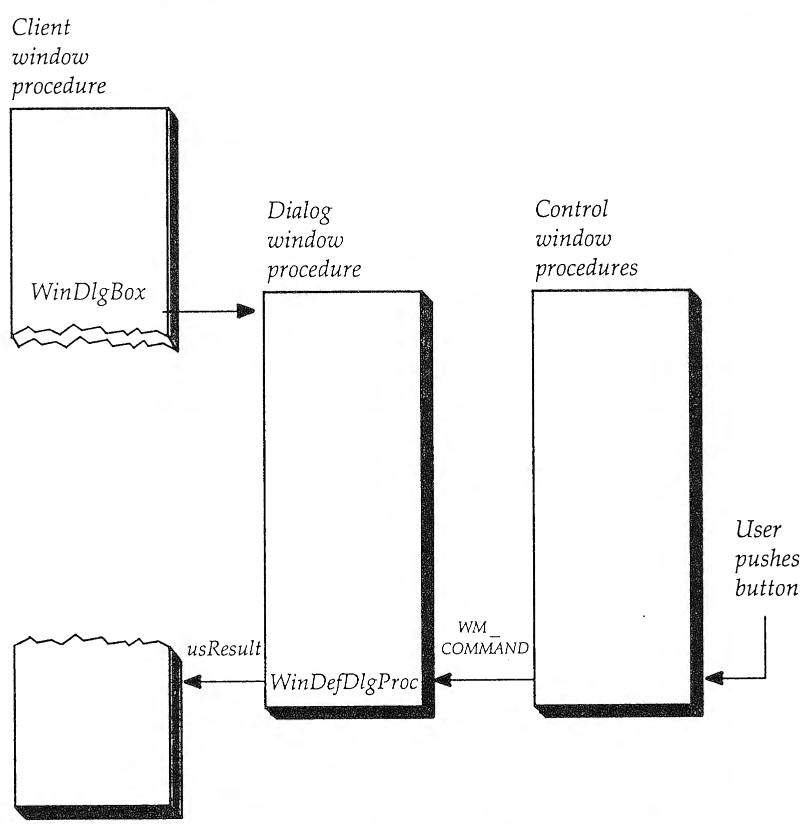


Figure 9-3. Message and control flow in DLGMSG

The ID number following DLGTEMPLATE identifies the dialog template. This is the ID normally used to access the dialog resource. In the current example we use it in WinDlgBox to specify which box we want to create.

After the DLGTEMPLATE statement there follow the usual BEGIN and END keywords (or curly brackets), delimiting the subsequent statements.

The DIALOG Keyword

The DIALOG keyword introduces the frame window that is the parent of the controls. The open and close quotes following DIALOG delimit the text that will appear in a title bar, if the frame window has one. In the DLGMSG example there is no title bar, so there's nothing between the quotes. The ID that follows the text is the ID of the frame window. We would use it if we wanted to refer to the frame window in particular (as opposed to the entire dialog data template), but this is not normally necessary. The next four numbers specify the position and size of the frame window. Again, the DIALOG statement is followed by BEGIN and END delimiters that will delimit statements specifying the controls that are its ownees.

Although they aren't shown here, there can be three more parameters to a DIALOG statement: the *style, framectl, data-definitions, presparams,* and *control-definitions.* We'll ignore these for the moment.

The Dialog Box Coordinate System

If the coordinate system used for dialog boxes were specified in pixels, the coordinates of the controls in the box would vary depending on the particular display adapter used in the computer system. To help make the coordinate system more device-independent, a different system is used for dialog boxes. Horizontal dimensions are specified in units that are one-quarter the width of an average system font character, and vertical dimensions are specified in units that are one-eighth the width of such a character. Characters are about twice as high as they are wide, so the vertical and horizontal dialog units are roughly the same size. Some resolution is lost with this system, since you can't specify the size or position of a control to pixel resolution; but this is seldom necessary in a dialog box. The X and Y positions and the height and width can range from 0 to a maximum of 65,535 dialog units.

(Incidentally, while the coordinates of controls in a dialog box are measured in dialog units, normal window positions are expressed in pixels. Should you need to convert from one system to the other, you can use the WinMapDlgPoints function.)

In the DLGMSG example the lower left corner of the box will be located 80 dialog units right of the left edge of the desktop, and 50 units above the bottom of the desktop. This translates into 20 character widths (80 divided by 4), and 6 1/4 character heights (50 divided by 8). The box will be 150 dialog units wide and 50 units high.

Specifying Controls

Keywords are used to specify the controls that will be placed in the dialog box. Here's a partial list:

Keyword	Control Placed in Dialog Box
LTEXT	Left-justified text
RTEXT	Right-justified text
CTEXT	Horizontally centered text
DEFPUSHBUTTON	Default push button (dark border)
PUSHBUTTON	Push button (light border)
RADIOBUTTON	Radio button
AUTORADIOBUTTON	Auto radio button
CHECKBOX	Check box
ENTRYFIELD	Entry field
LISTBOX	List box
GROUPBOX	Group box

These control specifications all have a similar format. After the keyword comes a *label,* which is the text to be displayed by the control (the word "Exit" in a push button, for example). Following the label is the ID of the control. Then come the X and Y positions, and the width and height. Finally a *style* parameter specifies styles, such as BS_PUSHBUTTON, SS_TEXT, SS_GROUPBOX, and so on, which are combined with the bitwise OR operator to specify the kind of control window.

In our example there are three controls in the dialog box: left-justified text and two push buttons. DEFPUSHBUTTON is similar to PUSHBUTTON, except that it creates a push button with a darker border. This is the default push button: the one we expect the user to push most of the time. Initially, pressing the ENTER key pushes this button.

Using an Editor to Create Dialog Boxes

The software development tools for PM include an editor, DLGBOX, that can be used to automate the creation of dialog boxes. This utility is a

draw-type program. It displays a border, representing the dialog box. You can size this border as desired by dragging its borders. Then you specify various controls, drag them to the proper location inside the border, and size them. A "Grid" menu selection makes it easier to line up controls that are visually related.

Output of the Dialog Editor

The good news is that you can use the dialog editor to automate the creation of your dialog boxes. The bad news is that the output from this program contains a great deal of unnecessary information, which makes it difficult to read. When we use the dialog editor to create the dialog box in DLGMSG, for example, we get something like this:

```
DLGINCLUDE 1 "MSGBOX.H"

DLGTEMPLATE ID_DLGBOX LOADONCALL MOVEABLE DISCARDABLE
BEGIN
 DIALOG "", ID_DLGBOX, 92, 55, 137, 65, FS_NOBYTEALIGN |
            FS_DLGBORDER | WS_VISIBLE | WS_CLIPSIBLINGS
            | WS_SAVEBITS
 BEGIN
  CONTROL "Are you sure you want to exit?", ID_MSG, 1, 53, 134, 8,
          WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP |
          WS_GROUP | WS_VISIBLE
  CONTROL "Yes", ID_YES, 27, 10, 38, 12, WC_BUTTON,
          BS_PUSHBUTTON |  BS_DEFAULT | WS_TABSTOP |
          WS_VISIBLE
  CONTROL "No", ID_NO, 79, 10, 38, 12, WC_BUTTON,
          BS_PUSHBUTTON |  WS_TABSTOP | WS_VISIBLE
 END
END
```

Actually we've cleaned up this output somewhat to make it more presentable, but it's still much more obscure than the hand-typed version. One problem is that all the default styles, such as LOADONCALL and WS_VISIBLE, are displayed. Also, all controls are defined by the keyword CONTROL, using class and style parameters to specify the kind of control. This is an alternative system, but not a more readable one. Compare this hand-typed statement:

```
DEFPUSHBUTTON "Yes", ID_YES, 27, 10, 38, 12
```

with the version generated by the dialog editor:

```
CONTROL "Yes", ID_YES, 27, 10, 38, 12, WC_BUTTON,
        BS_PUSHBUTTON |  BS_DEFAULT | WS_TABSTOP |
        WS_VISIBLE
```

In the dialog editor all controls are generated using the CONTROL keyword. This keyword is followed by the following parameters, separated by commas:

Parameter	Purpose
text	Title or label if appropriate
id	Control identifier
x	X coordinate of lower left corner (dialog units)
y	Y coordinate of lower left corner (dialog units)
width	Width of control (dialog units)
height	Height of control (dialog units)
class	Window class (WC_BUTTON, etc.)
style	Window style (BS_PUSHBUTTON, etc.)
data-definitions	Specifies control or presentation data

The *class* argument in our example, WC_BUTTON, describes the button class, just as it did when we created controls with WinCreateWindow in Chapter 6. The *style* parameter consists of several identifiers ORed together, again as in WinCreateWindow. BS_DEFAULT makes a default (dark border) button, and WS_TABSTOPS means the keyboard focus will stop at this control when it's cycled among the controls with the TAB key.

We don't use the last two parameters to CONTROL in our example.

As you can see, keywords like PUSHBUTTON and LTEXT are a shorthand approach to using CONTROL with the appropriate class and static identifiers, such as WC_BUTTON, WC_STATIC, SS_TEXT, and so forth.

Another approach to creating dialog templates is to use the dialog editor to create an initial example of the resource, and then modify this with a text editor to remove unnecessary identifiers and adjust the coordinates where necessary. This allows you to use the dialog editor where it's most helpful: in visually establishing the coordinates of the controls. At the same time you end up with a more readable .RC file. We'll follow this approach in the next example.

The .DLG File

The output of the dialog editor is a .DLG text file. This file must be made part of your resource file. There are two ways to do this. First you can keep the .DLG file separate and place an RCINCLUDE statement in your resource file. For example, to include the file DLGMSG.DLG, place the statement

```
RCINCLUDE dlgmsg.dlg
```

in your .RC resource file.

The second approach is to use your text editor to merge the text of the .DLG file into the .RC file.

Header Files

As we've seen before, a .H header file is used to contain the *#define* directives for the ID numbers used to identify the dialog template, dialog window, and control windows.

If you type the dialog resource into the resource file by hand, this poses no problems. The dialog editor, on the other hand, generates its own *#define*s and places them in a .H file. Currently, it does not like to see any statements other than *#define*s in the .H file. Thus you can't use an existing .H file. This is another problem with the dialog editor that we hope will be cleared up in future releases.

Simple Dialog Box Summary

Here are the steps necessary to create a simple dialog box:

1. Create a DLGTEMPLATE resource, by hand or with the dialog editor.
2. Add a dialog window procedure to your .C file.
3. Initiate the dialog box with WinDlgBox.
4. Process messages from the dialog box controls in the dialog window procedure.

Using WinDlgBox is the simplest way to invoke a dialog box. However, it's not very flexible. The application calls WinDlgBox, and the only information it receives from the resulting dialog box is the return value from this function. It's rather like a glorified message box. In the next section we'll examine a more flexible approach.

MORE COMPLEX DIALOG BOXES

When we use WinDlgBox, our client window procedure can't communicate with the dialog box between the time the box is created and the time it's destroyed. This is unfortunate if we want to initialize the dialog box after it's created, or read a value stored in a control. However, we can replace WinDlgBox with several other functions when we want this flexibility.

Our next example displays a text phrase, "Text in center of window", in the client window. There are two menu items: "Change" and "Exit". Selecting "Exit" terminates the program immediately. Selecting the "Change" item, on the other hand, invokes a dialog box, as shown in Figure 9-4.

This dialog box contains static text, an entry field, a group box, three radio buttons, and two push buttons. The static text describes the entry field; it reads "Text:". The entry field is initialized with the phrase originally displayed in the client window. The user can modify this phrase by interacting with the entry field. All or part of the phrase can be deleted by backspacing the cursor across it, or by selecting part of it and pressing the DEL key. Text can also be inserted at the cursor position, but only until the box is full. The contents of the entry field will be written to the screen when the box is dismissed by clicking on the "OK" button.

A group box is a border that provides a visual grouping of related elements, and has a title. The group box in our example has the title "Colors" and surrounds the three radio buttons. These radio buttons are labeled "Red", "Green", and "Blue". Clicking on one of them changes the color of the text written to the screen from red (the default selection) to green or blue. The color change will take effect when the "OK" button is pressed and the box is dismissed.

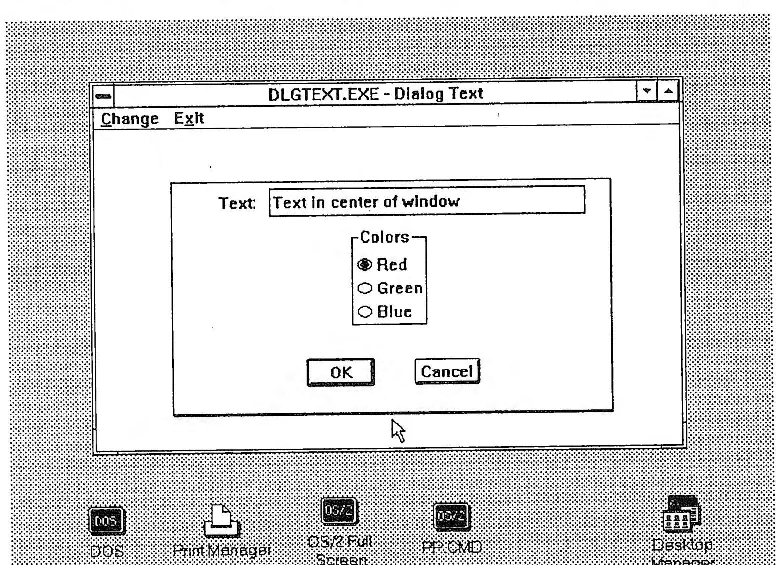


Figure 9-4. Output of the DLGTEXT program

Clicking on either of the two push buttons will dismiss the dialog box. Clicking the "OK" button will cause the text and the color entered into the dialog box controls to be displayed in the center of the application's window. Choosing the "Cancel" button or ESCAPE will cause this input to be forgotten; the text in the window will remain unchanged.

Here are the listings for DLGTEXT.C, DLGTEXT.H, DLGTEXT, DLGTEXT.DEF, DLGTEXT.RC, and DLGTEXT.DLG.

```
/* ---------------------------- */
/* DLGTEXT - Using a dialog box */
/* ---------------------------- */

#define INCL_WIN
#define INCL_GPI
#include <os2.h>

#include "dlgtext.h"

HWND hwndFrame;

USHORT cdecl main(void)
   {
   HAB hab;                              /* handle for anchor block */
   HMQ hmq;                              /* handle for message queue */
   QMSG qmsg;                            /* message queue element */
   HWND hwndClient;                      /* handles for windows */

                                         /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_MENU;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);            /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);      /* create message queue */

                                         /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                         /* create standard window */
   hwndFrame = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
                  szClientClass, " - Dialog Text", 0L, NULL,
                  ID_FRAMERC, &hwndClient);

                                         /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwndFrame);          /* destroy frame window */
   WinDestroyMsgQueue(hmq);              /* destroy message queue */
   WinTerminate(hab);                    /* terminate PM usage */

   return 0;
   }
```

```
                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   HPS hps;
   RECTL rcl;
   static CHAR szText[80] = "Text in center of window";
   static COLOR clr = CLR_RED;          /* initial color is red */
   HWND hwndDlg;
   static USHORT usCheckedButton = ID_RED;
   USHORT usResult;

   switch (msg)
      {
      case WM_COMMAND:
         switch (COMMANDMSG(&msg)->cmd)
            {
            case ID_NEWTEXT:             /* New text selected */

               hwndDlg = WinLoadDlg(
                  HWND_DESKTOP,           /* parent */
                  hwndFrame,              /* owner */
                  DlgProc,                /* dialog win proc */
                  NULL,                   /* dialog template in EXE */
                  ID_DLGBOX,              /* dialog template ID */
                  NULL);                  /* no params */

                                          /* set entry field text */
                  WinSetDlgItemText(hwndDlg, ID_TEXT, szText);
                                          /* check radio button */
                  WinSendDlgItemMsg(hwndDlg, usCheckedButton,
                     BM_SETCHECK, MPFROMSHORT(TRUE), MPFROMSHORT(NULL));

                                          /* wait for dlg box */
                usResult =  WinProcessDlg(hwndDlg);

                                          /* if not cancel button */
                if (usResult != DID_CANCEL)
                  {
                  switch (usResult)      /* record new color */
                     {
                     case ID_RED: clr = CLR_RED; break;
                     case ID_GREEN : clr = CLR_GREEN; break;
                     case ID_BLUE : clr = CLR_BLUE; break;
                     }
                                        /* query new text */
                  WinQueryDlgItemText(hwndDlg, ID_TEXT,
                     sizeof(szText), szText);

                  WinDestroyWindow(hwndDlg);    /* bye bye dlg win */

                  usCheckedButton = usResult;
                  WinInvalidateRect(hwnd, NULL, FALSE);   /* repaint */
                  }
               break;

            case ID_EXIT:                /* Exit selected */
```

```
                  WinPostMsg(hwnd, WM_QUIT, NULL, NULL);
                  break;
               }
            break;

         case WM_PAINT:
               hps = WinBeginPaint(hwnd, NULL, NULL);
               WinQueryWindowRect (hwnd, &rcl);
               WinDrawText (hps, -1, szText, &rcl, clr,
                  CLR_BACKGROUND, DT_CENTER | DT_VCENTER | DT_ERASERECT);
               WinEndPaint(hps);
               break;

         default:
            return(WinDefWindowProc(hwnd, msg, mp1, mp2));
            break;
         }

      return NULL;                          /* NULL / FALSE */
      }

                                            /* dialog window procedure */
   MRESULT EXPENTRY DlgProc(HWND hwnd, USHORT msg, MPARAM mp1,
                        MPARAM mp2)
      {
      static USHORT usResult = ID_RED;      /* initial color */
      RECTL rclScreen, rclDlg;

      switch(msg)
      {
      case WM_CONTROL:                      /* radio button pushed */
         switch SHORT1FROMMP(mp1)
               {
               case ID_RED:
               case ID_GREEN:
               case ID_BLUE:
                  usResult = SHORT1FROMMP(mp1);
               break;
               }
         break;

      case WM_COMMAND:                      /* button (ok/cancel) pushed */
            if (COMMANDMSG(&msg)->cmd == DID_CANCEL)
               usResult = DID_CANCEL;
            WinDismissDlg(hwnd, usResult);
            break;

      case WM_INITDLG:                      /* center dlg box on screen */
         WinQueryWindowRect(HWND_DESKTOP, &rclScreen);
         WinQueryWindowRect(hwnd, &rclDlg);
         WinSetWindowPos(hwnd, 0,
            (short)(rclScreen.xRight - (rclDlg.xRight - rclDlg.xLeft))/2,
            (short)(rclScreen.yTop - (rclDlg.yTop - rclDlg.yBottom))/2,
            0, 0, SWP_MOVE);
         break;

      default:
            return(WinDefDlgProc(hwnd, msg, mp1, mp2));
```

```
            break;
         }

      return NULL;                              /* NULL / FALSE */
      }
```

```
/* --------- */
/* DLGTEXT.H */
/* --------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);
MRESULT EXPENTRY DlgProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_NEWTEXT 101
#define ID_EXIT 102

#define ID_DLGBOX     200
#define ID_MSG        211
#define ID_TEXT       221
#define ID_COLORS     230
#define ID_RED        231
#define ID_GREEN      232
#define ID_BLUE       233
```

```
# -----------------
# DLGTEXT Make file
# -----------------

dlgtext.obj: dlgtext.c dlgtext.h
   cl -c -G2s -W3 -Zp dlgtext.c

dlgtext.res: dlgtext.rc dlgtext.h dlgtext.dlg
   rc -r dlgtext.rc

dlgtext.exe: dlgtext.obj dlgtext.def
   link /NOD dlgtext,,NUL,os2 slibce,dlgtext
   rc dlgtext.res dlgtext.exe

dlgtext.exe: dlgtext.res
   rc dlgtext.res
```

```
; -----------
; DLGTEXT.DEF
; -----------

NAME        DLGTEXT  WINDOWAPI

DESCRIPTION 'Using a dialog box'

PROTMODE
STACKSIZE   4096
```

```
/* --------- */
/* DLGTEXT.RC */
/* --------- */

#include <os2.h>
#include "dlgtext.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "~Change", ID_NEWTEXT
      MENUITEM "E~xit", ID_EXIT
   END

RCINCLUDE dlgtext.dlg
```

```
/* ----------- */
/* DLGTEXT.DLG */
/* ----------- */

DLGINCLUDE 1 "DLGTEXT.H"

DLGTEMPLATE ID_DLGBOX LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG "", ID_DLGBOX, 57, 34, 240, 96, FS_NOBYTEALIGN | FS_DLGBORDER |
                WS_VISIBLE | WS_CLIPSIBLINGS | WS_SAVEBITS
    BEGIN
        CONTROL "Text:", ID_MSG, 24, 85, 20, 8, WC_STATIC, SS_TEXT |
                DT_LEFT | DT_VCENTER | WS_GROUP | WS_VISIBLE
        CONTROL "", ID_TEXT, 54, 85, 172, 8, WC_ENTRYFIELD,
                ES_LEFT | ES_MARGIN | WS_TABSTOP | WS_VISIBLE
        CONTROL "Colors", ID_COLORS, 97, 36, 42, 42, WC_STATIC,
                SS_GROUPBOX | WS_GROUP | WS_VISIBLE
        CONTROL "Red", ID_RED, 100, 57, 32, 10, WC_BUTTON,
                BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
        CONTROL "Green", ID_GREEN, 100, 47, 37, 10, WC_BUTTON,
                BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
        CONTROL "Blue", ID_BLUE, 100, 37, 32, 10, WC_BUTTON,
                BS_AUTORADIOBUTTON | WS_TABSTOP | WS_VISIBLE
        CONTROL "OK", DID_OK, 72, 10, 38, 12, WC_BUTTON,
                BS_PUSHBUTTON | BS_DEFAULT | WS_GROUP | WS_TABSTOP |
                WS_VISIBLE
        CONTROL "Cancel", DID_CANCEL, 131, 10, 38, 12, WC_BUTTON,
                BS_PUSHBUTTON | WS_VISIBLE
    END
END
```

Creating a Dialog Window with WinLoadDlg

When the user selects the "Change" item from the menu, a WM_COMMAND message with a command value of ID_NEWTEXT is sent to the client window procedure. The client window procedure responds by executing WinLoadDlg. Like WinDlgBox, this function creates a

dialog box, based on the dialog template in the resource file. Unlike WinDlgBox, however, WinLoadDlg returns immediately; it doesn't wait for the dialog box to be dismissed.

Create Dialog Window

```
HWND WinLoadDlg(hwndParent, hwndOwner, pfnDlgProc, hmod, idDlg,
                pCreateParams)
HWND hwndParent       Parent window of dialog box
HWND hwndOwner        Owner window of dialog box
PFNWP pfnDlgProc      Dialog window procedure
HMODULE hmod          Resource module (NULL=.EXE file)
USHORT idDlg          ID of dialog window in resource file
PVOID pCreateParams   Dialog box specific control data

Returns: Dialog box handle, or NULL if unsuccessful
```

The arguments to this function are the same as those for WinDlgBox. The return value is the handle of the dialog box; this handle will be used in subsequent references to the dialog box.

The WM_INITDLG Message

Once the dialog box is created, WinLoadDlg causes a WM_INITDLG message to be sent to the dialog window procedure (*DlgProc,* in our example). WinLoadDlg returns as soon as WM_INITDLG has been sent. This message is the signal for the dialog window procedure to perform whatever initialization is necessary for the dialog box. In our example this initialization consists of positioning the dialog box in the exact center of the screen. If we didn't do this, it would position itself, not necessarily where we wanted it.

The message sent on initialization is one of the two main differences between dialog window procedures and client window procedures. Client window procedures use WM_CREATE, while dialog window procedures use WM_INITDLG. The other difference, which we already discussed, is that client window procedures use WinDefWindowProc for default processing, while dialog window procedures use WinDefDlgProc.

Positioning a Window with WinSetWindowPos

There are three steps to positioning the dialog box. First, we execute WinQueryWindowRect to find the dimensions of the screen. Second, we

execute this function again to find the dimensions of the dialog window. The third step requires a new function.

The WinSetWindowPos positions (or repositions) a window. It can also resize the window and change it in other ways.

Position a Window

```
BOOL WinSetWindowPos(hwnd, hwndInsertBehind, x, y, cx, cy, fs)
HWND hwnd                Window to be repositioned
HWND hwndInsertBehind    Z-axis ordering (HWND_TOP, etc.)
SHORT x                  X position of lower left corner
SHORT y                  Y position of lower left corner
SHORT cx                 Width
SHORT cy                 Height
USHORT fs                Positioning options (SWP_SIZE, etc.)

Returns: TRUE if successful, FALSE if error
```

The first parameter to this function is the window to be repositioned. The next parameter is *hwndInsertBehind,* which can be HWND_TOP, HWND_BOTTOM, or the handle of a window behind which this window will be placed. (See the section on WinCreateWindow in Chapter 4 for a discussion of Z-axis ordering.) The *x* and *y* arguments specify the coordinates of the lower left corner of the window, and the *cx* and *cy* arguments specify the width and height of the window. In the cached micro presentation space we're using, dimensions are measured in pixels.

The last argument to WinSetWindowPos, *fs,* can be one or more SWP_ identifiers, ORed together. These identifiers specify what action WinSetWindowPos will take and which parameters it will use. Here are common identifiers:

Identifier	**Meaning**
SWP_SIZE	Change size
SWP_MOVE	Change position
SWP_ZORDER	Change Z-axis ordering
SWP_SHOW	Show window when created
SWP_HIDE	Hide window when created
SWP_ACTIVATE	Activate, if top main window
SWP_DEACTIVATE	Deactivate, if window is active
SWP_MINIMIZE	Minimize
SWP_MAXIMIZE	Maximize

In our example we don't need to resize the window, only to position it, so we use SWP_MOVE. The new X position is the width of the screen less the width of the dialog window, divided by two. The new Y position is the height of the screen less the height of the dialog window, divided by two. Since the dialog window is only moved—not resized or reordered on the Z-axis—*cx, cy* can be set to 0, and *hwndInsertBehind* can be set to NULL.

Interacting with an Active Dialog Box

Once the dialog window has been created, WinLoadDlg returns. The dialog window is now active: The user can interact with it. At the same time our client window procedure is free to interact with it or with its various control windows. In our example the client window procedure first sends text to the entry field control, and then checks (that is, blackens the center of) one of the radio button controls. Because these interactions take place with the controls themselves, rather than with the dialog window, they do not need to be processed by the dialog window procedure, so there is no code in our application to handle them. Figure 9-5 shows the communications paths.

Setting Text in a Control Item

The client window procedure uses the WinSetDlgItemText function to initialize the text in the entry field control item in the dialog window.

Set Text in Dialog Control Item

```
SHORT WinSetDlgItemText(hwndDlg, idItem, pszText)
HWND hwndDlg        Dialog window handle
USHORT idItem       Identifier of control item
PSZ pszText         Text to be sent (0-terminated string)

Returns: TRUE if successful, FALSE if error
```

This function requires the handle of the dialog window, the ID number of the control item in the dialog window, and the text to be sent to the control.

WinSetDlgItemText is similar to another API, WinSetWindowText. WinSetDlgItemText requires the ID of the target window and the handle of the parent of this window. This is the usual situation with dialog window

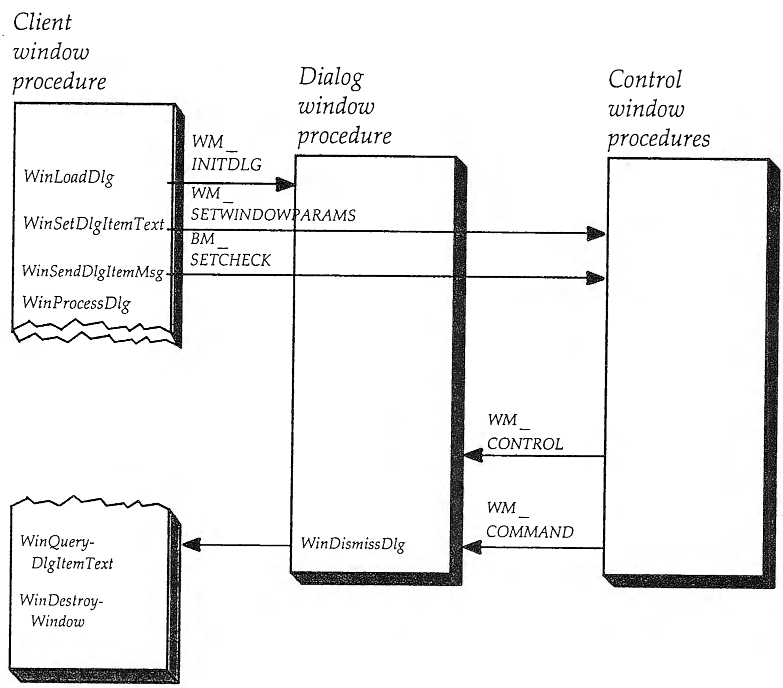


Figure 9-5. Message and control flow in DLGTEXT

controls. WinSetWindowText requires only the handle of the target window. Thus WinSetDlgItemText is equivalent to calling WinWindowFromID to get the handle of a child when the parent is known, and using this handle to call WinSetWindowText. WinSetDlgItemText can be used not only with dialog windows, but anytime you know a window's ID and parent, but not its handle.

A related function, WinSetDlgItemShort, can be used to send an integer to a dialog item and display it there as a text string. Another function, WinQueryDlgItemShort, reads a text string containing digits from a control item and returns the numerical value they represent. These functions make it easy to interpret numbers typed into entry fields by the user.

Sending Messages to Control Items

The client window procedure can send messages to control items in the dialog box. In DLGTEXT we send a BM_SETCHECK message to one of the radio buttons, telling it to check itself (blacken its middle). This message is sent with the WinSendDlgItemMsg function.

Send Message to Dialog Control Item

```
MRESULT WinSendDlgItemMsg(hwndDlg, idItem, msg, mp1, mp2)
HWND hwndDlg      Dialog window handle
USHORT idItem     Identifier of control item
USHORT msg        Message identifier
MPARAM mp1        Message parameter 1 (message dependent)
MPARAM mp2        Message parameter 2 (message dependent)

Returns: Result returned by control window to which message was sent
```

This function is similar to WinSendMsg, except that it sends a message to a child window of a known window, rather than to the window itself. Thus it requires one more parameter than WinSendMsg: the ID of the child window. Like WinSendMsg, this function does not return until the message has been processed by the recipient.

Instead of using WinSendDlgItemMsg, you could find the handle of the control window itself using WinWindowFromID, and send it the message with WinSendMessage. WinSendDlgItemMsg is useful not just with dialog windows, but anytime you want to send a message to a window whose parent and ID are known, but not its handle.

The first part of *mp1* in the BM_SETCHECK message is set to TRUE to check the button and to FALSE to uncheck a previously checked button. The *mp2* parameter is not used.

Waiting for the User with WinProcessDlg

In the previous example, DLGMSG, one function, WinDlgBox, created the dialog box, waited for it to be dismissed, and then destroyed it. In DLGTEXT these activities are divided among three separate functions. WinLoadDlg creates the dialog box, and another function, WinProcessDlg, waits for the user to dismiss it. (We'll find out soon how it's destroyed.) As we've seen, using this multistep process permits the application to interact with the dialog box while it is active.

Process Dialog Messages

```
USHORT WinProcessDlg(hwndDlg)
HWND hwndDlg      Dialog window handle

Returns: Value passed by WinDismissDlg
```

The result returned by WinProcessMsg is the same value returned by WinDismissDlg in the dialog window procedure. Let's see what the dialog window procedure does.

usResult in the Dialog Window Procedure

If the user pushes one of the radio buttons, the button will send a WM_CONTROL message to the dialog window procedure, with the first part of *mp1* indicating the ID of the button. The dialog window procedure then sets a variable *usResult* to this ID. This result will be returned by WinDismissDlg when the user dismisses the dialog box by clicking on the "OK" button.

On the other hand, if the user dismisses the dialog box by clicking on the "Cancel" button, WinDismissDlg returns the ID of this button, DID_CANCEL. In either case, executing WinDismissDlg causes the dialog window procedure to return with a value of *usResult.* The dialog window also vanishes from the screen.

usResult in the Client Window Procedure

When WinDismissDlg is executed in the dialog window procedure, WinProcessDlg returns in the client window procedure. Its return value is the same as that given to WinDismissDlg. This is assigned to *usResult* in the client window procedure (a different local variable than *usResult* in the dialog window procedure). The *usResult* value determines what the client window procedure will do next. If its value is DID_CANCEL, no action is taken. If it is not DID_CANCEL, the "OK" button must have been pushed. In this case the color value *clr* is set to the corresponding color value (CLR_BLUE if *usResult* is ID_BLUE, for example). This color value is used later in the WinDrawText function.

Gone but Not Forgotten

Besides the color selected by the user, the client window procedure needs to know what text the user typed into the entry field in the dialog box. But the dialog box has become invisible because WinDismissDlg was executed: It has vanished from the screen and the user can no longer interact with it. How can we find out anything about it? In fact, the dialog box still exists, and the state of all its controls is unaltered. So we can query the controls to ascertain the state of the dialog box when it was dismissed. (We'll see in a minute how to kill it off completely.)

Reading the Entry Field

To find out what text the user typed into the dialog box's entry field requires a new function, WinQueryDlgItemText.

Get Text from Dialog Control Item

```
USHORT WinQueryDlgItemText(hwndDlg, idItem, cchBufferMax, pchBuf)
HWND hwndDlg              Dialog box handle
USHORT idItem             ID of control item
SHORT cchBufferMax        Size of buffer
PSZ pchBuf                Buffer for text

Returns: Length of text returned, or 0 if error
```

This function takes the text from the control that has an ID of *idItem* and is located in the dialog window *hwndDlg*, and writes it into *pchBuff*, which is *cchBufferMax* bytes long. In our example this is the buffer *szText*.

Destroying the Dialog Box

If the dialog box still exists after WinProcessMsg has returned, how do we get rid of it completely? This is accomplished with the WinDestroyWindow function, which we first encountered in Chapter 4. This function frees the system resources being used by the dialog window. It also destroys the control windows that were the descendants of the dialog window.

We have used three APIs to accomplish what WinDlgBox did in the previous example. WinLoadDlg creates the dialog box, but returns immediately, so the client window procedure can do other processing, including communicating with the controls in the dialog box. WinProcessDlg waits for the user to dismiss the dialog box. After this function returns, the client window procedure can again communicate with the controls in the dialog box. Finally, WinDestroyWindow administers the coup de grace.

Once the dialog box is destroyed, DLGTEXT sets the *usCheckedButton* variable equal to *usResult*. It then causes the new text to be redrawn in the new color by invalidating the window, which causes the WM_PAINT message to be sent.

Dialog Box Summary

This example has covered most of the elements involved in using a full-scale dialog box. Here's a summary of the steps involved:

1. Create a DLGTEMPLATE resource, by hand or with the dialog editor.
2. Add a dialog window procedure to your .C file.

3. Create the dialog box with WinLoadDlg.
4. Set values in the controls in the dialog box with WinSetDlgItemText or similar APIs.
5. Process messages from the controls in the dialog box in the dialog window procedure.
6. Using WinProcessDlg, wait for the user to dismiss the box.
7. Get values from controls in box with WinQueryDlgItemText, and so on.
8. Destroy the dialog box with WinDestroyWindow.

Which Kind of Dialog Box?

Which approach to creating a dialog box should you use?

As we've seen, the message box is used when very simple choices are to be made. Only three or fewer buttons, with stock text such as "OK," "Yes," and "No" can be used, with one text field and an icon.

A more complex dialog box can be created with WinDlgBox. Any number of radio buttons, check boxes, text, and so on can be installed in it. However, the client window procedure must be content with a single return value from the dialog box. The user's interaction with the box must boil down to a single value.

When the client window procedure needs to initialize elements in the dialog box after it's created, or read more data from it than can be expressed in a single return value, then the dialog box can be created, processed, and destroyed separately with WinLoadDlg, WinProcessDlg, and WinDestroyWindow. This approach should also be used for modeless dialog boxes.

DIALOGS AND STANDARD WINDOWS

A dialog box, or dialog window, is very similar to a standard window. Both a standard window and a dialog window are really a *family of windows.* They are both based on a frame window, and both can be augmented with child windows such as system menus, sizing borders, and so on. Both can contain child windows in the form of controls, such as push buttons.

A standard window is usually created with WinCreateStdWindow. System menus and so on are associated with it using parameters to this function, and control windows can be added by executing WinCreateWindow, as we saw in Chapter 6. A dialog window, on the other hand, is usually created as a dialog template in a resource, and displayed with WinDlgBox or WinLoadDlg.

But these differences in approach mask the fact that standard windows and dialog windows are essentially the same. They are handled differently not because they are different kinds of windows, but because it's more convenient to do so. A dialog window usually owns a great many control windows. It's easier to create a dialog template resource to specify these controls than it is to create the dialog window with WinCreateStdWindow and execute WinCreateWindow repeatedly to create the controls. On the other hand, it's easier to create a single standard window with a few children, such as a system menu and title bar, using WinCreateStdWindow.

To demonstrate the similarity between a dialog and a standard window, we've concocted a program whose standard window is created as if it were a dialog window—using a resource and WinDlgBox.

The 15 Puzzle

The example is a *15 Puzzle.* This is an ancient puzzle that consists of 15 numbered blocks in a 4×4 framework. The framework can hold 16 blocks; but one square is empty. By repeatedly sliding a block into the empty square, the pattern of the blocks can be altered. Figure 9-6 shows some

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Normal

	15	14	13
12	11	10	9
8	7	6	5
4	3	2	1

Backwards

1	3	5	7
9	11	13	15
2	4	6	8
10	12	14	

Odd/Even

1	2	3	4
12	13	14	5
11		15	6
10	9	8	7

Spiral

Figure 9-6. Patterns in the 15 Puzzle

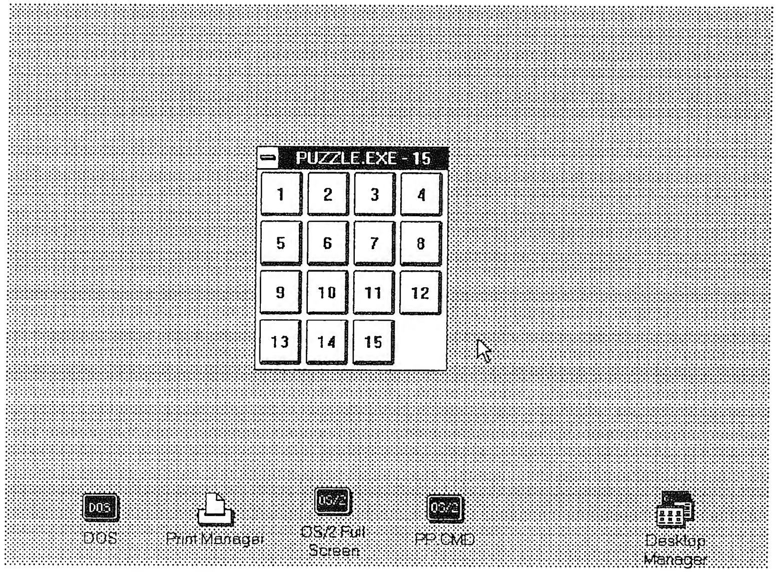


Figure 9-7. Output of the PUZZLE program

typical patterns.

The PUZZLE program appears as a standard window with a system menu and a title bar. Within the window are 15 controls: push buttons, numbered from 1 to 15. (A 16th button exists, but it is invisible.) Figure 9-7 shows the output of the program when it's first executed.

By clicking on a button adjacent to the vacant square, the user causes that button to "move" into the square. Actually the button doesn't move, it stays in the same place and its text is moved.

Trying to rearrange the 15 Puzzle into different patterns should keep you—or your children—busy for quite some time. You'll also find the program construction amusing. The listings for PUZZLE.C, PUZZLE.H, PUZZLE, PUZZLE.DEF, and PUZZLE.RC follow.

```
/* -------------------------------- */
/* PUZZLE - 15 puzzle using dlg box */
/* -------------------------------- */

#define INCL_WIN
#include <os2.h>

#include "puzzle.h"

USHORT cdecl main(void)
   {
```

```
   HAB hab;                                /* handle for anchor block */
   HMQ hmq;                                /* handle for message queue */
   QMSG qmsg;                              /* message queue element */
   HWND hwnd;                              /* handles for windows */

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);              /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);        /* create message queue */

                                           /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, 0L, 0);

   hwnd = WinLoadDlg( HWND_DESKTOP, HWND_DESKTOP, NULL,
             NULL, ID_PUZZLE, NULL);

                                           /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);                 /* destroy dialog window */
   WinDestroyMsgQueue(hmq);                /* destroy message queue */
   WinTerminate(hab);                      /* terminate PM usage */

   return 0;
   }

                                           /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                         MPARAM mp2)
   {
   static USHORT usEmpty = 44;             /* initial empty square (4,4) */
   CHAR szText[3];
   HPS hps;

   switch (msg)
      {
      case WM_COMMAND:
                                           /* check for valid move */
         if ((COMMANDMSG(&msg)->cmd + 1 == usEmpty) ||
             (COMMANDMSG(&msg)->cmd - 1 == usEmpty) ||
             (COMMANDMSG(&msg)->cmd + 10 == usEmpty) ||
             (COMMANDMSG(&msg)->cmd - 10 == usEmpty))
            {
                                           /* copy button's text */
            WinQueryDlgItemText(hwnd, COMMANDMSG(&msg)->cmd,
               sizeof(szText), szText);
            WinSetDlgItemText(hwnd, usEmpty, szText);
                                           /* un-hide old empty */
            WinShowWindow(WinWindowFromID(hwnd,usEmpty), TRUE);
            usEmpty = COMMANDMSG(&msg)->cmd;
                                           /* hide new empty */
            WinShowWindow(WinWindowFromID(hwnd,usEmpty), FALSE);
            }
         else                              /* invalid move */
            WinAlarm(HWND_DESKTOP, WA_NOTE);
         break;

      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
```

```
      GpiErase(hps);                    /* erase old button */
      WinEndPaint(hps);
      break;

   default:
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }

return NULL;                            /* NULL / FALSE */
}
```

```
/* -------- */
/* PUZZLE.H */
/* -------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAME 2
#define ID_PUZZLE 200
#define ID_41 41
#define ID_42 42
#define ID_43 43
#define ID_44 44
#define ID_31 31
#define ID_32 32
#define ID_33 33
#define ID_34 34
#define ID_21 21
#define ID_22 22
#define ID_23 23
#define ID_24 24
#define ID_11 11
#define ID_12 12
#define ID_13 13
#define ID_14 14
```

```
# ----------------
# PUZZLE Make file
# ----------------

puzzle.obj: puzzle.c puzzle.h
   cl -c -G2s -W3 -Zp puzzle.c

puzzle.res: puzzle.rc puzzle.h
   rc -r puzzle.rc

puzzle.exe: puzzle.obj puzzle.def
   link /NOD puzzle,,NUL,os2 slibce,puzzle
   rc puzzle.res puzzle.exe

puzzle.exe: puzzle.res
   rc puzzle.res
```

```
; ----------
; PUZZLE.DEF
; ----------

NAME        PUZZLE  WINDOWAPI

DESCRIPTION '15 Puzzle'

PROTMODE
STACKSIZE   4096
```

```
/* --------- */
/* PUZZLE.RC */
/* --------- */

#include <os2.h>
#include "puzzle.h"

WINDOWTEMPLATE ID_PUZZLE
   BEGIN
      FRAME " - 15", ID_FRAME, 100, 200, 106, 85,
         WS_VISIBLE | FS_BORDER,
         FCF_SYSMENU | FCF_TITLEBAR | FCF_TASKLIST
         BEGIN
            WINDOW "", FID_CLIENT, 1, 0, 106, 85, "Client Window",
               WS_VISIBLE
               BEGIN
                  PUSHBUTTON  "1", ID_11,  1, 64, 25, 20, BS_NOPOINTERFOCUS
                  PUSHBUTTON  "2", ID_12, 27, 64, 25, 20, BS_NOPOINTERFOCUS
                  PUSHBUTTON  "3", ID_13, 53, 64, 25, 20, BS_NOPOINTERFOCUS
                  PUSHBUTTON  "4", ID_14, 79, 64, 25, 20, BS_NOPOINTERFOCUS
                  PUSHBUTTON  "5", ID_21,  1, 43, 25, 20, BS_NOPOINTERFOCUS
                  PUSHBUTTON  "6", ID_22, 27, 43, 25, 20, BS_NOPOINTERFOCUS
                  PUSHBUTTON  "7", ID_23, 53, 43, 25, 20, BS_NOPOINTERFOCUS
                  PUSHBUTTON  "8", ID_24, 79, 43, 25, 20, BS_NOPOINTERFOCUS
                  PUSHBUTTON  "9", ID_31,  1, 22, 25, 20, BS_NOPOINTERFOCUS
                  PUSHBUTTON "10", ID_32, 27, 22, 25, 20, BS_NOPOINTERFOCUS
                  PUSHBUTTON "11", ID_33, 53, 22, 25, 20, BS_NOPOINTERFOCUS
                  PUSHBUTTON "12", ID_34, 79, 22, 25, 20, BS_NOPOINTERFOCUS
                  PUSHBUTTON "13", ID_41,  1,  1, 25, 20, BS_NOPOINTERFOCUS
                  PUSHBUTTON "14", ID_42, 27,  1, 25, 20, BS_NOPOINTERFOCUS
                  PUSHBUTTON "15", ID_43, 53,  1, 25, 20, BS_NOPOINTERFOCUS
                  CONTROL      "", ID_44, 79,  1, 25, 20, WC_BUTTON,
                                  BS_PUSHBUTTON | BS_NOPOINTERFOCUS
               END
         END
   END
```

The *main* Function

The *main* function has been altered in an unexpected way in this program. There is no WinCreateStdWindow function. Instead, the function WinLoadDlg, in conjunction with a dialog resource template, is used to create

the standard window and its ownees. Since the window template loaded by WinLoadDlg is used as a standard window, it is associated with a client window procedure rather than a dialog window procedure. In fact, there is no dialog window procedure.

The Resource File

The program's standard window is defined in the resource script file. There are no menus or other resources in the program, so the standard window is the only element in the .RC file.

Instead of DLGTEMPLATE, this resource uses WINDOWTEMPLATE. These two keywords have identical effects, but using the second one emphasizes that the window is being treated more like a standard window than a dialog window.

The Frame Window

Instead of the keyword DIALOG, we use FRAME. This creates a frame window just as DIALOG does, but again, using FRAME emphasizes that we're creating the frame window of a standard window.

Child Windows

If we had created the standard window with WinCreateStdWindow, we would have used identifiers like FCF_TITLEBAR in the *flCreateFlags* argument to add child windows to the frame window. Since we're creating it with WinDlgBox, we achieve the same effect by placing these identifiers in the window template (or dialog template, if you prefer). We use them to add a system menu (so we can terminate the program) and a title bar (so we can move the window).

Client Window

We also specify a client window using the keyword WINDOW. This client window will be a child and an ownee of the frame window, and is the same size as the frame. It's defined to be of "Client Window" class, and this class is associated with *ClientWinProc* using WinRegisterClass just as it would be for any other client window.

Control Windows

This client window is the parent and owner of the push button control windows. The resource template defines 16 such button controls, which are children and ownees of the client window. Of these, 15 are defined using the PUSHBUTTON keyword, and one is defined with CONTROL. The reason for this is that one button is created invisible. The default for PUSHBUTTON is visible, and this can't be overridden. To create an invisible push button, the CONTROL keyword must be used, since its default is not visible. The BS_PUSHBUTTON style is used with CONTROL to specify a push button.

All the push buttons are sized at 25×20, and positioned at the appropriate coordinates in the frame window to form a 4×4 matrix. The text is set to the numbers from 1 to 15 for the first 15 of the buttons. The last button is given no text; it doesn't need any since it's invisible.

The ID number of each button is derived from its position. For example, the button on the third line in the second column (initially "10") is given an ID of 32, as can be seen from the value of ID_32 in the .H file. This is shown in Figure 9-8.

Using ID numbers related to the button's position makes it easy to analyze whether a button is adjacent to an empty square, as we'll see in a moment.

You may have noticed that none of the buttons in PUZZLE has the dotted line that indicates keyboard focus. Such a dotted line, lingering on the last button to be clicked, would be distracting to the user. Accordingly, the keyboard focus is deactivated by applying the BS_NOPOINTER-FOCUS style on all the push buttons.

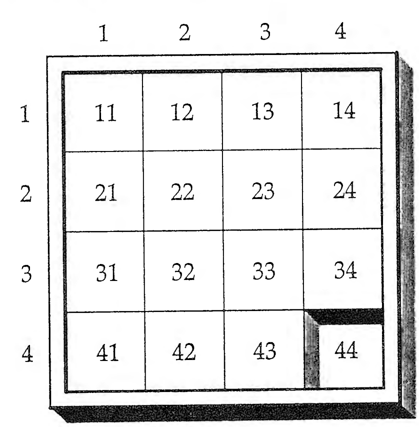


Figure 9-8. Numbering of button IDs in PUZZLE

The Client Window Procedure

When the user clicks on one of the buttons, it sends a WM_COMMAND message to its owner, the client window procedure. As in previous examples, the ID of the control sending the message is contained in the *COMMANDMSG(&msg)->cmd* message parameter.

The first job of the client window procedure on receiving a WM_COMMAND message is to see if the button was valid, that is, if the user clicked on a button adjacent to the empty square. If the selected button is to the left or right of the empty square, its ID will be 1 less or 1 greater than the ID of the empty square. If it's above or below, its ID will be 10 less or 10 greater. The program checks for these four cases, and if none is true, then it beeps, signaling that the user clicked on a button that can't be moved.

If the button was valid, it must be moved into the empty square. Or more precisely, it must appear to move into the empty square. There are two aspects to this. First the button's text ("7", or whatever) is transferred to the button occupying the empty square. Then the selected button must be made invisible, and the button in the new empty square must be made visible.

Transferring the text is carried out with WinQueryDlgItemText, which reads the text from the selected button into *szText*, and WinSetDlgItemText, which writes it from *szText* into the control whose ID number is stored in *usEmpty*. As we noted in the last example, these are useful functions even though we are not, strictly speaking, working with a dialog window.

The empty button is made visible and the selected button is made invisible using a new API: WinShowWindow.

Make Window Visible or Invisible

```
BOOL WinShowWindow(hwnd, fShow)
HWND   hwnd      Window handle
BOOL   fShow     TRUE=make window visible, FALSE=invisible

Returns: TRUE if successful, FALSE if error
```

Since WinShowWindow requires the handle of a window (the button) instead of its ID, the handle is obtained with WinWindowFromID.

WinShowWindow first makes visible the button whose ID is stored in *usEmpty* window. Then *usEmpty* is given the ID of the selected button. This new button, now stored in *usEmpty*, is made invisible.

Making a button invisible does not remove it from view, however. It gives the button's window procedure the invisible style, but does nothing with the image already drawn on the screen. However, making a button invisible means that the area behind it, on the client window, becomes invalid. When part of the client window becomes invalid, it receives a WM_PAINT message. This tells WinBeginPaint to update the invalid part of the window, so GpiErase is applied to just the invalid part of the window—the square we want to make invisible. The empty square is erased and vanishes.

The techniques used in this example show how window templates can be used to create a whole family of windows using only one API call. We could have achieved the same effect with repeated calls to WinCreateWindow, but the template approach is far easier to implement.

PART II FINALE

With this chapter we conclude our discussion of the user interface. You've learned all its major components: windows, controls, menus, and dialog boxes. You've also learned about the message-based architecture of PM applications and about other features of PM programming such as resources.

You now know enough to create full-scale PM applications with sophisticated user interfaces. You don't need to invent your own user-interface functions or purchase specialized user-interface libraries; it's all built into the system. With the user interface out of the way, you can concentrate on the heart of your application: its "real work."

Do you still want to go back to programming in MS-DOS? The PUZZLE program requires only a few dozen lines of code to create and control 15 push buttons. Even when the resource is included in the line count, the program is short for what it does. It's also conceptually quite elegant, with the hard work being done in a resource. The same simplicity and elegance are apparent in other PM programs. Once you get the hang of programming in PM, you can accomplish far more, with less trouble, than in any traditional procedure-based programming environment.

But the user interface isn't all that PM provides. There is also a powerful graphics capability that permits pictures to be created, stored, moved from place to place, and manipulated in very sophisticated ways. We cover this next, in Part III. There are also important PM programming considerations that are not directly related to the user interface, such as writing applications in a multitasking environment, and exchanging data between applications. We'll discuss these topics in Part IV.

III

GRAPHICS AND TEXT

10

LINES, ARCS, AND MARKERS

Chapters 10 through 14 are concerned with the graphics APIs that PM makes available to application programmers. These APIs provide amazingly powerful graphics capabilities.

In this chapter we'll show how to use three simple graphics primitives—lines, arcs, and markers—display them on the screen, and print them on the printer. Chapter 11 covers characters (another graphics primitive) and fonts. Chapter 12 examines areas, paths, regions, and bitmaps, and introduces clipping. Chapter 13 explores one of PM's most powerful graphics capabilities: *retained graphics,* in which graphics elements are combined in segments and chains that can be stored, "played back" to reproduce an image, and manipulated to create animation and other effects. Chapter 13 also explores transformations, in which graphic images are translated from one "space" to another and can change shape and size during the process. Chapter 14 covers several advanced graphics topics.

GRAPHICS BASICS

Before we show program examples of graphics primitives, let's pause for a quick overview of some of the graphics ideas we'll be using in this and other chapters. These are presentation space, device context, primitives, attributes, and the coordinate system.

Presentation Space

In Chapter 5 we touched on the subjects of presentation space (PS) and device context. We introduced the *cached micro PS,* which we used thereafter for all example programs that required graphics output. We showed examples in which this PS was obtained with WinBeginPaint and also with WinGetPS. Now let's look more closely at the idea of presentation space.

A presentation space is an area of memory that contains specifications for a drawing environment, such as the current color and drawing position, and the color tables and fonts in use. Each application that intends to create graphics must request a presentation space from PM, and refer to this PS whenever it draws anything.

Device Independent

One of the important points about presentation spaces is that they are *device independent.* No matter what output device your graphics output is ultimately destined for—screen, printer, or whatever—the PS is the same. Thus in most instances you can write your application without regard to physical devices.

Types of Presentation Space

There are four types of presentation space: normal, micro, cached micro, and AVIO. AVIO is a special-purpose text-only PS that we'll mention in the chapter on characters and fonts. The other three PSs are all used for normal text and graphics. Which one to use is determined by a trade-off between capability on the one hand, and memory space and speed on the other.

The *normal PS* handles all graphics operations, but occupies the most memory and is the slowest. It can be used for retained graphics, and it can be associated with one device context, and then later reassociated with another device context. (We'll discuss device contexts next.) An application obtains a normal PS by issuing GpiCreatePS and releases it with Gpi-DestroyPS.

A *micro PS* does not permit certain kinds of graphics operations. Most importantly, you can't use retained graphics. Also, when a micro PS is created, it is associated with a particular device context and can't be reassociated with another one. If you don't need to use retained graphics or reassociate the device context, you should use a micro PS, which is faster and requires less memory than a normal PS. A micro PS can be used to

output images to windows on the screen or to devices like printers and plotters. An application obtains and releases a micro PS with the same functions used for a normal PS.

A *cached micro PS* is used only for output to windows on the screen. It shares the limitations of the micro PS. PM keeps a number of cached micro PSs available in a memory storage area (or cache, hence the name), so they are immediately available. They take the least memory and are the fastest, but cannot be used for output to printers or any other device besides the screen. The cached micro PS is obtained with WinBeginPaint or WinGetPS and released with WinEndPaint or WinReleasePS.

Choosing a PS

You should use the type of PS that makes the fewest demands on system resources and still has the power to do what you want. In this and the next two chapters we'll mostly use the cached micro PS. We'll use the micro PS in the section on printing in this chapter and the normal PS when we use retained graphics in Chapter 13.

Device Context

The *device context* provides a connection between the PS, which is device independent, and a physical device.

In the examples so far, we have not needed to specify a device context directly. This is because the cached micro PS obtained with WinBeginPaint or WinGetPS is already associated with a window device context. When we use micro and normal presentation spaces, we'll need to obtain device contexts with the WinOpenWindowDC or DevOpenDC functions. Once obtained, the device context must then be *associated* with the presentation space using either another function, GpiAssociate, or arguments to Gpi-CreatePS. This links the presentation space with a particular device. Figure 10-1 shows the relation between presentation space and device context.

Graphics Primitives

Graphics pictures in PM are constructed from graphics *primitives.* These are the fundamental visual elements of PM. There are six such primitives: lines, arcs, markers, characters, areas, and images. We'll look at the first three in this chapter.

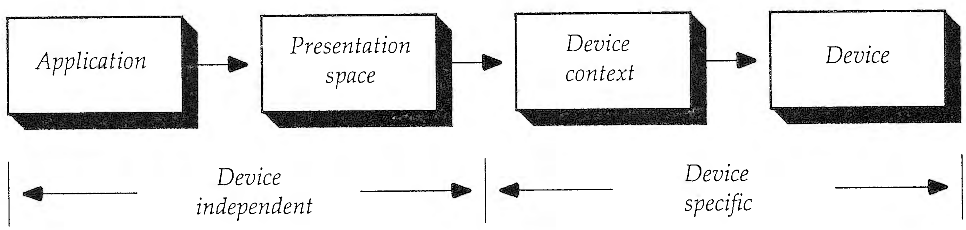


Figure 10-1. Presentation space and device context

Attributes

Attributes are characteristics that describe primitives. For instance, *color* is an attribute that can be applied to any of the primitives: there can be a blue line or a yellow area.

Many attributes apply only to certain primitives. For instance, *line type* applies only to lines and arcs, not to markers or areas. *Pattern,* on the other hand, applies to areas, but not to lines or markers.

We'll cover the attributes that apply to a particular primitive as we describe the primitive. Color will be introduced gradually throughout this and the next chapter.

The Coordinate System

As we mentioned in previous chapters, the origin of the coordinate system used for drawing in PM is at the lower left corner of the window. X coordinates increase to the right, and Y coordinates increase upward.

By default, dimensions for the cached micro PS are in pixels. That's what we've used in our examples to date. However, they can be changed to millimeters, inches, or other units, as we'll see later in this chapter.

Points

Many of the Gpi functions use points as parameters. A point is commonly represented by a structure with two members, defined this way in the PMGPI.H header file:

```
typedef struct _POINTL  {     /* ptl */
   LONG x;                    /* x coordinate of point */
   LONG y;                    /* y coordinate of point */
} POINTL;
```

Current Position

One important point (often of type POINTL) is the *current position.* This is the starting point for drawing lines or other graphics primitives. A function that draws a line, for example, does not specify the starting point of the line segment, only the end point. The starting point is assumed to be the current position. When a function finishes a drawing operation, the current position usually remains at the end point of whatever was drawn.

The current position can be changed using the GpiMove function. Moving the current position has no visual effect in itself; it will only be evident when another function, such as GpiLine, draws something starting at the current position. The current position is an attribute of a particular presentation space.

With these preliminaries out of the way, let's do some drawing.

LINE PRIMITIVES

In this section we'll demonstrate line primitives. We'll also introduce two important attributes of the line primitive: line type and color.

Three APIs can be used to draw line primitives. GpiLine draws a single line segment from one point to another. GpiPolyLine draws a series of line segments through points specified in an array. GpiBox creates a rectangle. In our first example we'll use all three of these functions to create a line graph, complete with border and grid lines.

A Graph Example

Here are the listings for LINE.C, LINE.H, LINE, and LINE.DEF.

```
/* ------------------- */
/* LINE - A line graph */
/* ------------------- */
```

```
#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "line.h"

USHORT cdecl main(void)
   {
   HAB hab;                             /* handle for anchor block */
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd, hwndClient;               /* handles for windows */

                                        /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);     /* create message queue */

                                        /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                        /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Line", 0L, NULL, 0, &hwndClient);

                                        /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);              /* destroy frame window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */

   return 0;
   }

                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   HPS hps;
   POINTL ptl;
   static POINTL aptlGraph[NGRAPHS][NPOINTS] = {
    {{5, 170}, {105, 140}, {205, 200}, {305, 180}, {405, 230}, {505, 220}},
    {{5, 20}, {105, 15}, {205, 60}, {305, 100}, {405, 110}, {505, 300}},
    {{5, 50}, {105, 20}, {205, 80}, {305, 60}, {405, 30}, {505, 10}},
    {{5, 100}, {105, 90}, {205, 85}, {305, 80}, {405, 75}, {505, 70}}};
   int i;

   switch (msg)
      {
      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
                                        /* draw grid */
```

```
        for (i = ORIGIN + DGRIDY; i < MAXGRIDY; i += DGRIDY)
           {
           ptl.x = ORIGIN;
           ptl.y = i;
           GpiMove(hps, &ptl);
           ptl.x = MAXGRIDX;
           GpiLine(hps, &ptl);
           }
        for (i = ORIGIN + DGRIDX; i < MAXGRIDX; i += DGRIDX)
           {
           ptl.y = ORIGIN;
           ptl.x = i;
           GpiMove(hps, &ptl);
           ptl.y = MAXGRIDY;
           GpiLine(hps, &ptl);
           }
                                         /* draw graphs */
        for (i = 0; i < NGRAPHS; i++)
           {
           GpiMove(hps, &aptlGraph[i][0]);
           GpiPolyLine(hps, (LONG)NPOINTS - 1, &aptlGraph[i][1]);
           }
                                         /* draw box */
        ptl.x = ptl.y = ORIGIN;
        GpiMove(hps, &ptl);
        ptl.x = MAXGRIDX;
        ptl.y = MAXGRIDY;
        GpiBox(hps, DRO_OUTLINE, &ptl, 0L, 0L);

        WinEndPaint(hps);
        break;

     case WM_ERASEBACKGROUND:
        return TRUE;                     /* erase background */
        break;

     default:
        return(WinDefWindowProc(hwnd, msg, mp1, mp2));
        break;
     }
  return NULL;                           /* NULL / FALSE */
  }
```

```
/* ------ */
/* LINE.H */
/* ------ */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ORIGIN 5
#define NGRAPHS 4
#define NPOINTS 6
#define MAXGRIDY 305
#define DGRIDY 50
#define MAXGRIDX 505
#define DGRIDX 100
```

```
# ---------------
# LINE Make file
# ---------------

line.obj: line.c line.h
   cl -c -G2s -W3 -Zp line.c

line.exe: line.obj line.def
   link /NOD /CO line,,NUL,os2 slibce,line
```

```
; --------
; LINE.DEF
; --------

NAME        LINE  WINDOWAPI

DESCRIPTION 'Draw a line graph'

PROTMODE
STACKSIZE   4096
```

There are no menus, dialog boxes, icons, or other niceties in this program, so the listing is simple and allows us to focus on the graphics functions. We'll use this same basic program in the next two examples, changing only the client window procedure.

In the LINE example we plot lines for four sets of data, as shown in Figure 10-2. The lines might represent gross sales for four regions over a six-month period.

A two-dimensional array of type POINTL contains the points to be plotted. There are NGRAPHS (4) sets of points, each of which will be plotted as a *polyline,* that is, the sequence of joined line segments created by GpiPolyLine. Each of the polylines contains NPOINTS (6) points. These constants are defined in LINE.H.

The GpiMove Function

The graph is drawn when the WM_PAINT message is received. The program first draws the sets of horizontal and vertical grid lines. In the first *for* loop, the horizontal lines start at ORIGIN and go to MAXGRIDX. In the second loop the vertical lines start at ORIGIN and run to MAXGRIDY. The ORIGIN constant has a value of 5, which is enough to separate the graph visually from the left and bottom edges of the window.

For each grid line, GpiMove first moves the current position to the start of the line.

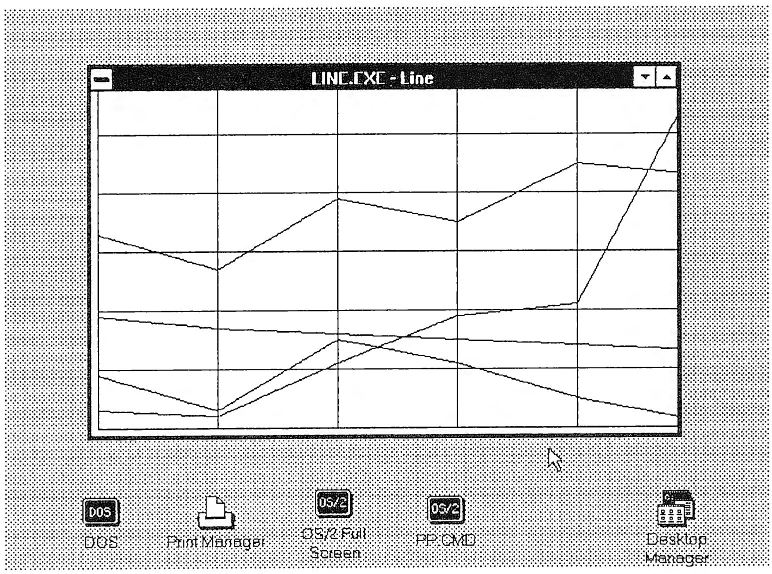


Figure 10-2. Output of the LINE program

Move Current Position

```
BOOL GpiMove(hps, pptl)
HPS hps                    Handle to presentation space
PPOINTL pptl               New position

Returns: GPI_OK if successful, GPI_ERROR if error
```

The two arguments to this function are the handle to the presentation space (used with almost all Gpi functions) and the point to which the current position should be moved.

Return value constants like GPI_OK are given in PMGPI.H, as are prototypes for all the Gpi functions. This file must be *#include*d in any program that uses Gpi functions. GPI_OK indicates the operation was successful. We don't check for errors in our example programs, but it's a good idea in a more serious program. If a Gpi function returns with GPI_ERROR, the specific error code can be discovered with WinGetLast-Error and additional information about the error can be recovered with WinGetErrorInfo.

The GpiLine Function

Once the starting point of a grid line has been set by GpiMove, the GpiLine function draws the line. The end points are at MAXGRIDX in the first loop and MAXGRIDY in the second.

Draw Line from Current Position to Specified Point

```
LONG GpiLine(hps, pptl)
HPS hps                   Handle to presentation space
PPOINTL pptl              Point to which line is drawn

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

The first argument to GpiLine is the ubiquitous presentation space handle. The next argument is the point to which the line should be drawn. GPI_HITS is returned when a "hit" takes place; we'll discuss hit-testing in Chapter 14.

The GpiPolyLine Function

GpiPolyLine is used to draw the polylines—the lines that connect the data points. The starting data point is the current position. The other data points are stored in an array.

Draw Line from Current Position to Multiple Points

```
LONG GpiPolyLine(hps, cptl, aptl)
HPS hps                   Handle to presentation space
LONG cptl                 Number of points in array
PPOINTL aptl              Array of points

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

There are two arguments to GpiPolyLine following the PS handle. The *cptl* argument is the number of points in the array. (This is the total number of points minus 1, since the current position is the starting point.) The *aptl* argument is a pointer to the array of POINTL structures holding the points.

The polylines are drawn using a *for* loop. Each pass through the loop draws one polyline. First GpiMove is used to set the current position to the start of the polyline on the left edge of the box. Then GpiPolyLine draws a series of line segments, starting at the current position and connecting the points for each polyline. When GpiPolyLine is finished, the current position remains at the end point of the last segment, on the right edge of the graph.

The GpiBox Function

Once the polylines are drawn, GpiBox draws a border around the entire graph. The lower left corner of the box is at (ORIGIN, ORIGIN). GpiMove moves the current position to this point. GpiBox then draws the box starting at this point and going to the upper right corner at (MAXGRIDX, MAXGRIDY).

Draw Box

```
LONG GpiBox(hps, cmdControl, pptl, lHRound, lVRound)
HPS hps                 Handle to presentation space
LONG cmdControl         DRO_FILL, DRO_OUTLINE, etc.
PPOINTL pptl            Opposite corner of box
LONG lHRound            For corner rounding
LONG lVRound            For corner rounding

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

The first argument is the PS handle, and the second indicates whether the box should be filled, have its outline drawn, or both. The possibilities are

Identifier	Fill/outline Action
DRO_FILL	Fills the interior
DRO_OUTLINE	Draws the outline
DRO_OUTLINEFILL	Draws the outline and fills the interior

We'll discuss fill in the next chapter, when we talk about areas. In the current example this argument is set to DRO_OUTLINE. (DRO stands for "drawing order.")

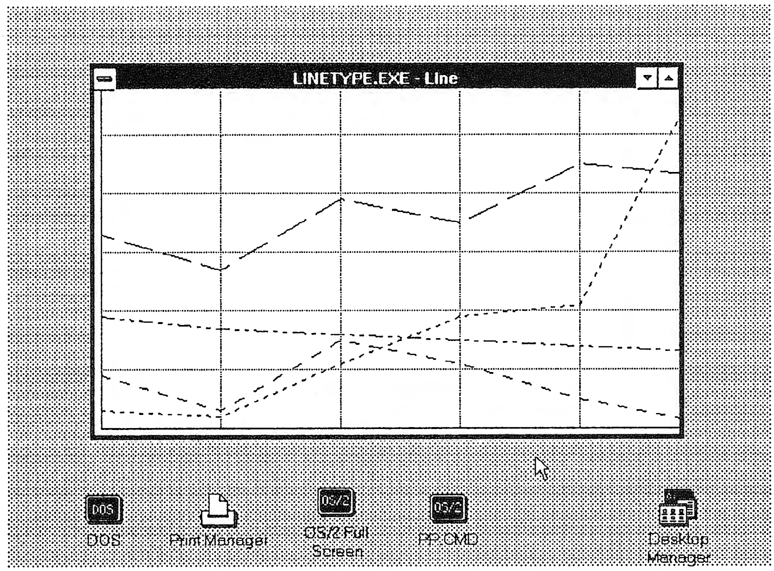


Figure 10-3. Output of the LINETYPE program

The third argument is the second corner of the box. (The first corner is the current position.) The last two arguments are the lengths of the horizontal and vertical axes of ellipses used to round the corners of the box. We use square corners, so these arguments are set to 0.

Line Type

One of the attributes a line may possess is *line type.* This attribute determines whether the line is solid, or is drawn with various combinations of dots or dashes.

Our next example draws the same graph as before, but uses a different line type for each polyline, so they can be more easily distinguished. The output from this program, LINETYPE, is shown in Figure 10-3.

Here's the listing for the client window procedure for LINETYPE. The other files and the *main* function are the same as those in LINE.

```
                                          /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                     MPARAM mp2)
   {
   HPS hps;
   POINTL ptl;
   static POINTL aptlGraph[NGRAPHS] [NPOINTS] = {
    {{5, 170}, {105, 140}, {205, 200}, {305, 180}, {405, 230}, {505, 220}},
```

```
 {{5, 20}, {105, 15}, {205, 60}, {305, 100}, {405, 110}, {505, 300}},
 {{5, 50}, {105, 20}, {205, 80}, {305, 60}, {405, 30}, {505, 10}},
 {{5, 100}, {105, 90}, {205, 85}, {305, 80}, {405, 75}, {505, 70}}};
static LONG aflLineType[NGRAPHS] =
   {
   LINETYPE_LONGDASH,
   LINETYPE_DOT,
   LINETYPE_SHORTDASH,
   LINETYPE_DASHDOUBLEDOT
   };
int i;

switch (msg)
   {
   case WM_PAINT:
      hps = WinBeginPaint(hwnd, NULL, NULL);
                                    /* draw grid */
      GpiSetLineType(hps, LINETYPE_ALTERNATE);   /* set line type */
      for (i = ORIGIN + DGRIDY; i < MAXGRIDY; i+= DGRIDY)
         {
         ptl.x = ORIGIN;
         ptl.y = i;
         GpiMove(hps, &ptl);
         ptl.x = MAXGRIDX;
         GpiLine(hps, &ptl);
         }
      for (i = ORIGIN + DGRIDX; i < MAXGRIDX; i += DGRIDX)
         {
         ptl.y = ORIGIN;
         ptl.x = i;
         GpiMove(hps, &ptl);
         ptl.y = MAXGRIDY;
         GpiLine(hps, &ptl);
         }
                                    /* draw graphs */
      for (i = 0; i < NGRAPHS; i++)
         {
         GpiSetLineType(hps, aflLineType[i]);   /* set line type */
         GpiMove(hps, &aptlGraph[i][0]);
         GpiPolyLine(hps, (long)NPOINTS - 1, &aptlGraph[i][1]);
         }
                                    /* draw box */
      ptl.x = ptl.y = ORIGIN;
      GpiMove(hps, &ptl);
      ptl.x = MAXGRIDX;
      ptl.y = MAXGRIDY;
      GpiSetLineType(hps, LINETYPE_SOLID);   /* set line type */
      GpiBox(hps, DRO_OUTLINE, &ptl, 0L, 0L);

      WinEndPaint(hps);
      break;

   case WM_ERASEBACKGROUND:
      return TRUE;                  /* erase background */
      break;

   default:
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }
return NULL;                        /* NULL / FALSE */
}
```

The GpiSetLineType function is used to set the line type.

Set Line Type

```
BOOL GpiSetLineType(hps, flLineType)
HPS hps           Handle to presentation space
LONG flLineType   Line type

Returns: GPI_OK if successful and GPI_ERROR if error
```

The possible values for *flLineType* are shown in Figure 10-4.

The LINETYPE_ALTERNATE identifier causes the pixels along the line to be alternately turned on and off, thus producing the effect of a lighter color (gray if the line color is black).

After GpiSetLineType is executed, all lines and arcs will be drawn with the new type, until the type is changed again. In our example GpiSetLineType is first used to set the type of the grid lines to LINETYPE_ALTERNATE. This lightens their appearance compared with the polylines and the surrounding box. The line types that will be used to draw the polylines are stored in the array *aLineType*. As the program cycles through the *for* loop, it uses a new line type for each polyline.

LINETYPE_DOT

LINETYPE_SHORTDASH

LINETYPE_DASHDOT

LINETYPE_DOUBLEDOT

LINETYPE_LONGDASH

LINETYPE_DASHDOUBLEDOT

LINETYPE_SOLID

LINETYPE_INVISIBLE

Figure 10-4. Line types

Color

Next we'll modify the client window procedure of our example to draw each polyline in a different color, rather than in a different type. Here's the client window procedure for LINECLR. The other files and *main* are the same as those in LINE.

```
                                       /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   HPS hps;
   POINTL ptl;
   static POINTL aptlGraph[NGRAPHS] [NPOINTS] = {
    {{5, 170}, {105, 140}, {205, 200}, {305, 180}, {405, 230}, {505, 220}},
    {{5, 20}, {105, 15}, {205, 60}, {305, 100}, {405, 110}, {505, 300}},
    {{5, 50}, {105, 20}, {205, 80}, {305, 60}, {405, 30}, {505, 10}},
    {{5, 100}, {105, 90}, {205, 85}, {305, 80}, {405, 75}, {505, 70}}};

   static COLOR aLineClr[NGRAPHS] =
      {
      CLR_RED,
      CLR_GREEN,
      CLR_BLUE,
      CLR_BROWN
      };

   int i;

   switch (msg)
      {
      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);

                                       /* draw grid */
         GpiSetColor(hps, CLR_YELLOW);    /* set color */
         for (i = ORIGIN + DGRIDY; i < MAXGRIDY; i+= DGRIDY)
            {
            ptl.x = ORIGIN;
            ptl.y = i;
            GpiMove(hps, &ptl);
            ptl.x = MAXGRIDX;
            GpiLine(hps, &ptl);
            }
         for (i = ORIGIN + DGRIDX; i < MAXGRIDX; i += DGRIDX)
            {
            ptl.y = ORIGIN;
            ptl.x = i;
            GpiMove(hps, &ptl);
            ptl.y = MAXGRIDY;
            GpiLine(hps, &ptl);
            }

                                       /* draw graphs */
         for (i = 0; i < NGRAPHS; i++)
```

```
        {
        GpiSetColor(hps, aLineClr[i]);   /* set color */
        GpiMove(hps, &aptlGraph[i][0]);
        GpiPolyLine(hps, (long)NPOINTS - 1, &aptlGraph[i][1]);
        }
                                       /* draw box */
      ptl.x = ptl.y = ORIGIN;
      GpiMove(hps, &ptl);
      ptl.x = MAXGRIDX;
      ptl.y = MAXGRIDY;
      GpiSetColor(hps, CLR_BLACK);   /* set color */
      GpiBox(hps, DRO_OUTLINE, &ptl, 0L, 0L);
      WinEndPaint(hps);
      break;

   case WM_ERASEBACKGROUND:
      return TRUE;                     /* erase background */
      break;

   default:
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }

return NULL;                           /* NULL / FALSE */
}
```

The GpiSetColor Function

The default color, which we've used until now, is black. To set the colors of the polylines—and the grid lines—we use the GpiSetColor function.

Set Foreground Color for All Primitives

```
BOOL GpiSetColor(hps, clr)
HPS hps             Handle to presentation space
COLOR clr           Color value

Returns: GPI_OK if successful, GPI_ERROR if error
```

The *clr* argument to this function is the color to be set. If the default logical color table is used, as in this example, this argument is a color index value with one of the following values:

Identifier	Color
CLR_FALSE	All color planes=0
CLR_TRUE	All color planes=1
CLR_DEFAULT	Same as CLR_NEUTRAL
CLR_WHITE	White
CLR_BLACK	Black
CLR_BACKGROUND	Default background (used by GpiErase)
CLR_BLUE	Blue
CLR_RED	Red
CLR_PINK	Magenta
CLR_GREEN	Green
CLR_CYAN	Cyan
CLR_YELLOW	Yellow
CLR_NEUTRAL	Default text color
CLR_DARKGRAY	Dark gray
CLR_DARKBLUE	Dark blue
CLR_DARKRED	Dark red
CLR_DARKPINK	Dark magenta
CLR_DARKGREEN	Dark green
CLR_DARKCYAN	Dark cyan
CLR_BROWN	Brown (dark yellow)
CLR_PALEGRAY	Light gray

Note that in PM parlance magenta is—possibly for brevity—called "pink"; a technical misnomer, since it's a combination of red and blue, not red and white.

Lines and arcs have only a foreground color, but—as we'll see when we discuss markers—some primitives have both foreground and background colors. GpiSetColor sets only the foreground color. After this function is executed, the foreground color of all primitives will be drawn in the specified color, until a new color is set.

Color Tables

The *logical color table* relates index values to RGB color values. All colors can be reproduced by combining different amounts of red, green, and blue light. White is maximum amounts of these three colors mixed together. Black is the absence of any color. Red is maximum red intensity, but no green or blue. Yellow is red and green, but no blue, and so on.

An RGB value is a 32-bit number specifying the amounts of red, green, and blue that yield a particular color. The upper eight bits aren't used. The next eight bits specify the amount of red, the next eight bits the amount of green, and the lower eight bits the amount of blue. Since each pair of hex digits represents eight bits, we can show the position of each color in a hex constant as 0x00RRGGBB.

Here are the first few entries in the default logical color table:

Index	RGB Value	Color
0	0x00FFFFFF	Device background color (white)
1	0x000000FF	Blue
2	0x00FF0000	Red
3	0x00FF00FF	Magenta
4	0x0000FF00	Green
5	0x0000FFFF	Cyan
6	0x00FFFF00	Yellow
7	0x00000000	Black

For windows on the screen the default device background color is white, but it can be changed from the control panel (or with the WinSetSysColors API). For printers it's the paper color.

The logical color table specifies the colors available to an application. But an application can output graphics to different devices, and each device may be capable of generating only certain colors. The colors for each device are stored in a *physical color table.* PM makes the closest possible matchup between the color chosen by the application from the logical color table, and the colors actually available from the physical color table. Logical color tables are important in understanding mix modes, which we'll mention in the section on markers later in this chapter and explore more thoroughly in Chapter 12.

An application can query a physical color table to determine what colors really are available. It can also query and modify the logical color table. We won't explore these possibilities here.

ARC PRIMITIVES

An arc is a curved line. The most usual form of arc is an ellipse (or a section of an ellipse), and the most commonly used ellipse is the circle. Arcs are classified in PM based on which API is used to construct them. The possibilities for ellipses are the *full arc*, the *partial arc*, and the *three-point arc*. PM

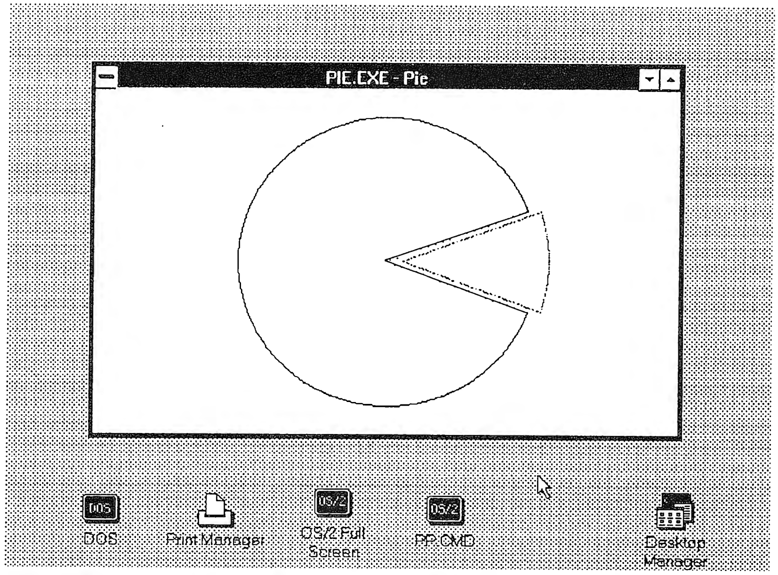


Figure 10-5. Output of the PIE program

also provides APIs to construct two kinds of arc that are not based on ellipses: the *fillet* and the *spline*. We'll provide examples of full and partial arcs, and mention the other three types (an example in Chapter 13 uses splines).

Partial Arcs

The partial arc is a part of an ellipse, one that subtends less than 360 degrees. Our example program, PIE, demonstrates this kind of arc by drawing a simple pie chart. For clarity, we've made it an *exploded* pie chart; that is, one in which some pieces are moved away from the others. Our example has two slices: a big one and a little one, as shown in Figure 10-5. The small slice might represent the proportion of a typical government budget not consumed by bureaucratic overhead.

Here are the listings for PIE.C, PIE.H, PIE, and PIE.DEF:

```
/* ---------------------- */
/* PIE - Draw a pie chart */
/* ---------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>
```

```
#include <stdlib.h>

#include "pie.h"

USHORT cdecl main(void)
   {
   HAB hab;                               /* handle for anchor block */
   HMQ hmq;                               /* handle for message queue */
   QMSG qmsg;                             /* message queue element */
   HWND hwnd, hwndClient;                 /* handles for windows */

                                          /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);             /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);       /* create message queue */

                                          /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                          /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Pie", 0L, NULL, 0, &hwndClient);

                                          /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);                /* destroy frame window */
   WinDestroyMsgQueue(hmq);               /* destroy message queue */
   WinTerminate(hab);                     /* terminate PM usage */

   return 0;
   }

                                          /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   HPS hps;
   static POINTL ptlCenter;
   static FIXED fxMult;
   static SIZEL sizl = {0, 0};
   switch (msg)
      {
      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
         GpiSetPS(hps, &sizl, PU_LOENGLISH);
                                          /* draw pie */
         GpiMove (hps, &ptlCenter);
         GpiSetColor(hps, CLR_DARKGREEN);
         GpiPartialArc(hps, &ptlCenter, fxMult, MAKEFIXED(20,0),
            MAKEFIXED(320,0));
         GpiLine(hps, &ptlCenter);
                                          /* draw slice */
         ptlCenter.x += FIXEDINT(fxMult) / 10;
         GpiMove(hps, &ptlCenter);
         GpiSetColor(hps, CLR_RED);
```

```
        GpiPartialArc(hps, &ptlCenter, fxMult, MAKEFIXED(340,0),
           MAKEFIXED(40,0));
        GpiLine(hps, &ptlCenter);

        WinEndPaint(hps);
        break;

     case WM_SIZE:                     /* window resized - resize pie */
        ptlCenter.x = SHORT1FROMMP(mp2) / 2;
        ptlCenter.y = SHORT2FROMMP(mp2) / 2;
        hps = WinGetPS(hwnd);
        GpiSetPS(hps, &sizl, PU_LOENGLISH);
        GpiConvert(hps, CVTC_DEVICE, CVTC_PAGE, 1L, &ptlCenter);
        fxMult = MAKEFIXED(min(min(ptlCenter.x, ptlCenter.y) * .85, 255),
                               0);
        WinReleasePS(hps);
        break;

     case WM_ERASEBACKGROUND:
        return TRUE;                   /* erase background */
        break;

     default:
        return(WinDefWindowProc(hwnd, msg, mp1, mp2));
        break;
     }

  return NULL;                         /* NULL / FALSE */
  }
```

```
/* ----- */
/* PIE.H */
/* ----- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);
```

```
# -------------
# PIE Make file
# -------------

pie.obj: pie.c pie.h
   cl -c -G2s -W3 -Zp pie.c

pie.exe: pie.obj pie.def
   link /NOD pie,,NUL,os2 slibce,pie
```

```
; -------
; PIE.DEF
; -------

NAME         PIE      WINDOWAPI

DESCRIPTION 'Draw a pie chart'

PROTMODE
STACKSIZE    4096
```

The GpiPartialArc Function

The function that draws a partial arc, which is what we want for the slices of a pie chart, is the appropriately named GpiPartialArc.

Draw a Partial Arc

```
LONG GpiPartialArc(hps, pptl, fxMultiplier, fxStartAngle,
                   fxSweepAngle)
HPS hps                    Handle to presentation space
PPOINTL pptl               Center of arc
FIXED fxMultiplier         Size multiplier (1 to 255)
FIXED fxStartAngle         Start angle in degrees
FIXED fxSweepAngle         Sweep angle in degrees

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

The argument following the PS handle is a POINTL structure holding the point that is the center of the arc. In our example this point, *ptlCenter*, is set on the arrival of a WM_SIZE message. This message returns the width and height of the window in the first and second parts of *mp1*. The center of the arc is set to half these dimensions to center it in the window.

The pie is drawn whenever a WM_PAINT message is received. An oddity—actually a convenience—of GpiPartialArc is that it not only draws the arc itself, but also one of the lines connecting the center of the arc with one of its ends. In a typical pie chart this eliminates drawing any other lines, since each slice draws the line separating it from the next slice. In our exploded example, however, we must connect the open ends of the slices. GpiLine draws the line from the open end of the arc, which is where the current position remains when the arc is finished, back to the center. This is shown in Figure 10-6.

The center of the arc is moved one tenth of the size of *fxMult* to the right before the second slice is drawn, to separate it from the first slice.

The third argument to GpiPartialArc is *fxMultiplier*. This argument is multiplied by the initial size of the arc to determine its final size. The default for the initial size is 1 unit; we use this default in this example. (We'll see in the next example how the arc can be given an initial size—and shape—other than the default.) The *fxMultiplier* argument is expressed as a value from 1 through 255, so the multiplier allows a circle with a radius between 1 and 255 units. The value used depends on the size of the window, as we'll see.

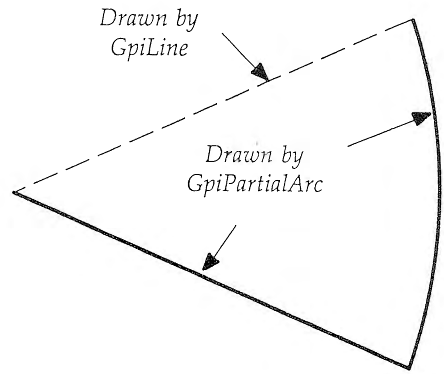


Figure 10-6. Completing a pie slice

The FIXED Data Type

The FIXED data type used for the last three arguments to GpiPartialArc requires some explanation. The FIXED type represents a number with a decimal point (or more accurately, a binary point). In this regard it's like type *float* in C, except that the position of the decimal point in *float* is determined by the exponent part of the variable, while in FIXED it is (as you may have guessed) fixed. Specifically, the decimal point is always located between bit 15 and bit 16 in a type LONG variable, as shown in Figure 10-7.

The low half of the variable represents the number below the decimal point and is type USHORT. The high half represents the number above the decimal point, and—since it carries the sign of the entire number—is type SHORT. Thus 2.0 is represented as 0x00020000 and –2.0 as 0xFFFE0000. The decimal fraction 2.5 would be 0x00028000, 2.25 would be 0x00024000, and so on.

The FIXED data type allows the representation of numbers with fractional parts, without the size and complexity of floating-point numbers. They're a good match for the kind of numbers used in graphics, which may have a small fractional part, but need not cover the huge magnitude of floating-point numbers.

Setting the Presentation Space

This example makes use of another PM feature: *page units.* The page unit is the unit used to measure the presentation space. Until now we have used

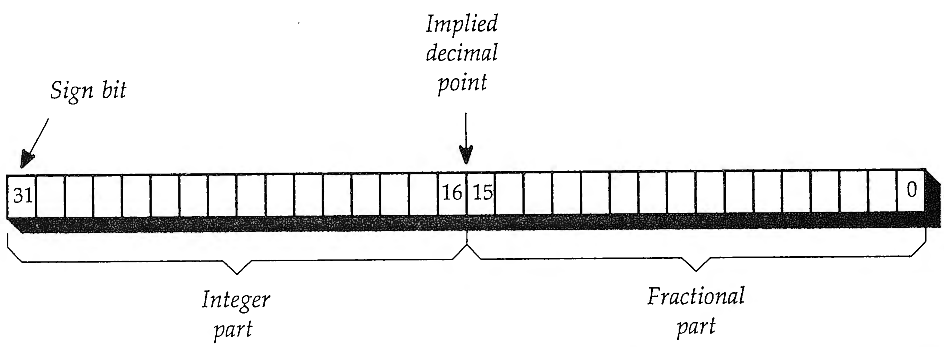


Figure 10-7. The FIXED data type

pixels, which are the default in the cached micro PS, as page units. However, the introduction of arcs provides the motivation to demonstrate a different approach. The reason this is necessary lies in a pixel's *aspect ratio.*

A display screen typically has an aspect ratio of 4 to 3. If it is 12 inches wide, for example, it will be 9 inches high. In VGA 640x480 mode each pixel has this same ratio, since 640/480 is 4/3 (1.333). In this case the physical pixels on the screen have the same height and width, and we say they are *square.* A graphics image created with square pixels has little distortion.

In other graphics modes, however, the pixels may not be square. In EGA 640x350 mode the ratio of height to width is 1.828. If a graphics figure based on pixel dimensions is drawn in this mode, it will appear to be squashed vertically, since there are too few pixels in the vertical dimension. Other graphics modes, and printers, may also have non-square pixels.

To avoid distortion when displaying or printing graphics images, so that circles appear as circles and not as ellipses, we need to use different units for specifying graphics primitives: inches or millimeters. If we specify, for example, a circle with a diameter of four inches, the display driver will automatically calculate how many pixels to use to make the circle four inches high and four inches wide. (PM doesn't know exactly how big to make screen images, since physical screen sizes vary, but it will correctly proportion them.) Images on printers and plotters will also be drawn in the correct size and proportion.

When the WM_PAINT message is received, WinBeginPaint is used in the usual way to obtain a cached micro PS. At this point pixels are the measurement unit. The function GpiSetPS is used to specify new units for the presentation space.

Set Presentation Space Page Size and Units

```
BOOL GpiSetPS(hps, psizl, flOptions)
HPS hps              Handle to presentation space
PSIZEL  psizl        Size of presentation page
ULONG flOptions      PS options (page units: PU_LOMETRIC, etc.)

Returns: GPI_OK if successful, GPI_ERROR if error
```

The *psizl* argument specifies the size of the presentation page. It's a pointer to a SIZEL structure, which specifies the dimensions of a rectangle. This structure is defined this way in PMGPI.H:

```
typedef struct _SIZEL    {     /* sizl */
   LONG cx;                    /* width of rectangle */
   LONG cy;                    /* height of rectangle */
   } SIZEL;
```

The page size is the size of the presentation space itself. If (0,0) is used for this argument, the presentation space will be given the same page size as the device to which it is writing. This is what we do in the example.

The *flOptions* argument specifies the page unit for the presentation space. Here are the options:

Identifier	**Unit Set to**
PU_ARBITRARY	Pixels initially, can be changed
PU_HIENGLISH	0.001 inch
PU_HIMETRIC	0.01 millimeter
PU_LOENGLISH	0.01 inch
PU_LOMETRIC	0.1 millimeter
PU_PELS	Pixels ("pels" to IBMers)
PU_TWIPS	Twip = a twentieth of a point (1/1440 inch)

We use 0.01-inch units, so we set the final argument to low English, specified by PU_LOENGLISH. (The *floptions* argument can also be used to specify how coordinates are stored in memory.)

Converting Coordinate Spaces

When we work with English or metric units, we must be careful to convert any pixel values to these same units. For example, the WM_SIZE message returns the window dimensions in pixels, so we need to convert these to

low English units to position the center of the circle. The function Gpi-Convert replaces the values for one or more points whose coordinates are expressed in one system, with the corresponding values in another system.

Convert Points Between Coordinate Spaces

```
BOOL GpiConvert(hps, lSrc, lTarg, cPoints, aptl)
HPS hps              Handle to presentation space
LONG lSrc            Source coordinate space
LONG lTarg           Target coordinate space
LONG cPoints         Number of points
PPOINTL aptl         Array of points

Returns: GPI_OK if successful, GPI_ERROR if error
```

The *lSrc* and *lTarg* arguments specify the source and target coordinate systems. The possibilities are

Identifier	**Type of Coordinate Space**
CVTC_DEFAULTPAGE	Page space before viewing transform
CVTC_DEVICE	Device space
CVTC_MODEL	Model space
CVTC_PAGE	Page space after viewing transform
CVTC_WORLD	World coordinates

We'll cover the concept of coordinate spaces and transformations in Chapter 13. Here, the device space is in pixels, and the page space is in low English, so *lSrc* is CVTC_DEVICE, and *lTarg* is CVTC_PAGE. The Gpi-Convert function can convert a whole array of points with one call. We want to convert only one point, so *cPoints,* the number of points in the array, is set to 1, and the address of the point, *&ptlCenter,* is used instead of an array, as the value of *aptl.*

Converting to FIXED

The *fxMultiplier* argument to GpiPartialArc, which we use to size the pie, is calculated on receipt of the WM_SIZE message. The pie should be smaller than the smallest dimension of the window. This corresponds to the smaller of *ptlCenter.x* and *ptlCenter.y.* The C *min* library function selects the smaller

of these, and 10 is subtracted from the result to provide a border around the pie. The *fxMultiplier* argument to GpiPartialArc can't be larger than 255, so we use the *min* function again to select 255 if the window has become too large.

The MAKEFIXED macro converts a whole number plus a fractional part to type FIXED. The whole number is the result of the *min* function just described, and the fractional part is 0.

Full Arcs

Full arcs are closed ellipses or circles, ones that subtend a full 360 degrees. They are generated by the GpiFullArc function. Closed arcs are typically used to represent car tires, the sun, happy faces, and similar graphics objects.

Create a Full Arc

```
LONG GpiFullArc(hps, flFlags, fxMultiplier)
HPS hps                  Handle to presentation space
LONG flFlags             DRO_FILL, DRO_OUTLINE, etc.
FIXED fxMultiplier       Size multiplier (1 to 255)

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

The possible values for *flFlags* are the same as those for GpiBox: DRO_FILL, DRO_OUTLINE, and DRO_OUTLINEFILL. The size multiplier is similar to that used in GpiPartialArc.

Changing Circles to Ellipses

By default, GpiFullArc (and, as we've seen, GpiPartialArc) draws circles. But when a circular object like a car tire or a coin is seen from an angle, it becomes an ellipse. How do we change circles to ellipses? This requires another function, GpiSetArcParams. This function sets four parameters that determine the shape and orientation of an arc. Once this function has been executed, any arcs drawn by GpiFullArc, or by the other arc APIs, will all have the new shape and orientation.

Set Arc Parameters

```
BOOL GpiSetArcParams(hps, parcp)
HPS hps                     Handle to presentation space
PARCPARAMS parcp            Arc parameters structure

Returns: GPI_OK if successful, GPI_ERROR if error
```

The second argument to this function points to an ARCPARAMS structure, which is defined in PMGPI.H like this:

```
typedef struct _ARCPARAMS {  /* arcp */
   LONG lP;                  /* horizontal scaling factor */
   LONG lQ;                  /* vertical scaling factor */
   LONG lR;                  /* horizontal shear factor */
   LONG lS;                  /* vertical shear factor */
} ARCPARAMS;
```

The four members of this structure are commonly referred to as P, Q, R, and S. For the algebraically inclined, a point (x, y) on the original circle is transformed into a point (x′, y′) on an ellipse with the equations

```
x' = x*P + y*R
y' = x*S + y*Q
```

Increasing P makes the circle wider, and increasing Q makes it higher. Increasing R moves the top of the circle to the right and the bottom to the left, and increasing S moves the right side of the circle up and the left side down.

The best way to get a feeling for these parameters is to experiment with them. The following example makes this easy: the parameters can be adjusted using menu items on the action bar, and the resulting ellipse is displayed. For variety this example uses low metric (0.1 mm) units instead of low English.

The program requires an .RC file to handle the menu resource. Here are ARC.C, ARC.H, ARC, ARC.DEF, and ARC.RC:

```
/* -------------------------- */
/* ARC - Arcs drawing program */
/* -------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>
#include <string.h>
```

```
#include "arc.h"

USHORT cdecl main(void)
   {
   HAB hab;                              /* handle for anchor block */
   HMQ hmq;                              /* handle for message queue */
   QMSG qmsg;                            /* message queue element */
   HWND hwnd, hwndClient;                /* handles for windows */

                                         /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_MENU;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);            /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);      /* create message queue */

                                         /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                         /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
            szClientClass, " - Arc", 0L, NULL, ID_FRAMERC, &hwndClient);

                                         /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);               /* destroy frame window */
   WinDestroyMsgQueue(hmq);              /* destroy message queue */
   WinTerminate(hab);                    /* terminate PM usage */

   return 0;
   }

                                         /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                     MPARAM mp2)
   {
   HPS hps;
   static ARCPARAMS arcp = {1, 1, 0, 0};
   static POINTL ptlCenter;
   static FIXED fxMult = MAKEFIXED(100, 0);
   static SIZEL sizl = {0, 0};
   static RECTL rcl = { 0, 0, 0, 0};
   static CHAR szText[80];
   switch (msg)
      {
      case WM_COMMAND:
         switch (COMMANDMSG(&msg) -> cmd)   /* new arc params */
            {
            case ID_PPLUS: arcp.lP++; break;
            case ID_PMINUS: arcp.lP--; break;
            case ID_QPLUS: arcp.lQ++; break;
            case ID_QMINUS: arcp.lQ--; break;
            case ID_RPLUS: arcp.lR++; break;
            case ID_RMINUS: arcp.lR--; break;
            case ID_SPLUS: arcp.lS++; break;
```

```
            case ID_SMINUS: arcp.lS--; break;
            case ID_MULTPLUS:
               if (fxMult < MAKEFIXED(246, 0))
                  fxMult += MAKEFIXED(10, 0);
               break;
            case ID_MULTMINUS:
               if (fxMult > MAKEFIXED(10, 0))
                  fxMult -= MAKEFIXED(10, 0);
               break;
            }
         WinInvalidateRect(hwnd, NULL, FALSE);   /* redraw */
         break;

      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
         GpiErase(hps);
         sprintf(szText, "P=%ld, Q=%ld, R=%ld, S=%ld, Multiplier=%d",
            arcp.lP, arcp.lQ, arcp.lR, arcp.lS, FIXEDINT(fxMult));
         WinDrawText(hps, -1, szText, &rcl, 0L, 0L, DT_TOP | DT_LEFT |
            DT_TEXTATTRS);
         GpiSetPS(hps, &sizl, PU_LOMETRIC);
         GpiMove (hps, &ptlCenter);
         GpiSetArcParams(hps, &arcp);  /* set arc params */
         GpiFullArc(hps, DRO_OUTLINE, fxMult);   /* draw arc */
         WinEndPaint(hps);
         break;

      case WM_SIZE:
         rcl.xRight = SHORT1FROMMP(mp2);
         rcl.yTop = SHORT2FROMMP(mp2);
         ptlCenter.x = SHORT1FROMMP(mp2) / 2;
         ptlCenter.y = SHORT2FROMMP(mp2) / 2;
         hps = WinGetPS(hwnd);
         GpiSetPS(hps, &sizl, PU_LOMETRIC);
         GpiConvert(hps, CVTC_DEVICE, CVTC_PAGE, 1L, &ptlCenter);
         WinReleasePS(hps);
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                        /* NULL / FALSE */
   }
```

```
/* ----- */
/* ARC.H */
/* ----- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1

#define ID_PPLUS  101
#define ID_PMINUS 102
#define ID_QPLUS  103
```

```
#define ID_QMINUS 104
#define ID_RPLUS  105
#define ID_RMINUS 106
#define ID_SPLUS  107
#define ID_SMINUS 109
#define ID_MULTPLUS  111
#define ID_MULTMINUS 112
```

```
# -------------
# ARC Make file
# -------------

arc.obj: arc.c arc.h
   cl -c -G2s -W3 -Zp arc.c

arc.res: arc.rc arc.h
   rc -r arc.rc

arc.exe: arc.obj arc.def
   link /NOD arc,,NUL,os2 slibce,arc
   rc arc.res

arc.exe: arc.res
   rc arc.res
```

```
; -------
; ARC.DEF
; -------

NAME         ARC       WINDOWAPI

DESCRIPTION 'Draw arcs'

PROTMODE
STACKSIZE   4096
```

```
/* ------ */
/* ARC.RC */
/* ------ */

#include <os2.h>
#include "arc.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "P+", ID_PPLUS
      MENUITEM "P-", ID_PMINUS
      MENUITEM "Q+", ID_QPLUS
      MENUITEM "Q-", ID_QMINUS
      MENUITEM "R+", ID_RPLUS
      MENUITEM "R-", ID_RMINUS
      MENUITEM "S+", ID_SPLUS
      MENUITEM "S-", ID_SMINUS
      MENUITEM "Multiplier+", ID_MULTPLUS
      MENUITEM "Multiplier-", ID_MULTMINUS
   END
```

Items on the action bar permit P, Q, R, and S to be increased or decreased. The argument *fxMult,* used in GpiFullArc to size the entire ellipse, can also be adjusted from the action bar. Figure 10-8 shows how increasing S changes the shape of the ellipse.

When P=Q=1 and R=S=0, the ellipse is a circle. These are the default values, which is why GpiFullArc draws circles in the absence of GpiSetArcParams.

Three-point Arcs, Fillets, and Splines

PM provides API functions for drawing several other kinds of curves. These are shown in Figure 10-9.

The *three-point arc* is an ellipse specified by three points on its circumference, rather than by its center and a multiplier. The P, Q, R, and S parameters determine the shape of the three-point arc, and GpiPointArc draws it from the current position through the second point to the third point.

The *fillet* is an arc that is drawn tangent to (parallel and touching) two straight control lines at their end points. These control lines are not displayed. The points that determine the control lines are given as arguments to GpiPolyFillet. Fillets are commonly used to curve one line smoothly into

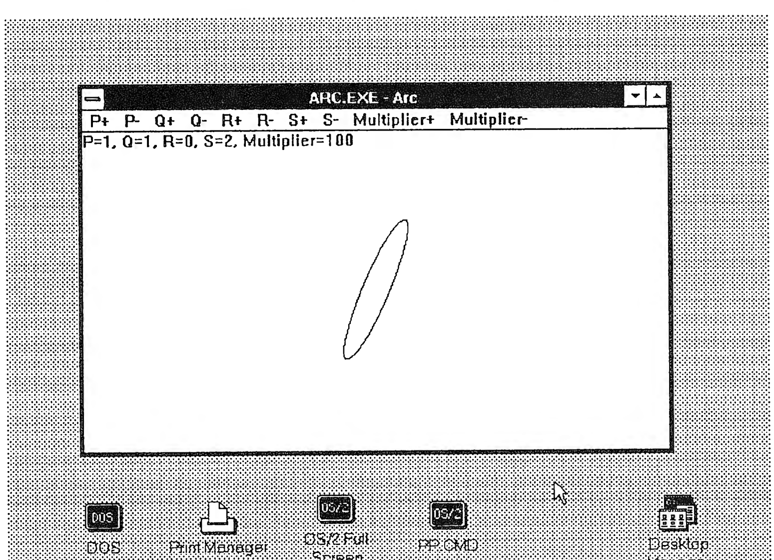


Figure 10-8. Output of the ARC program

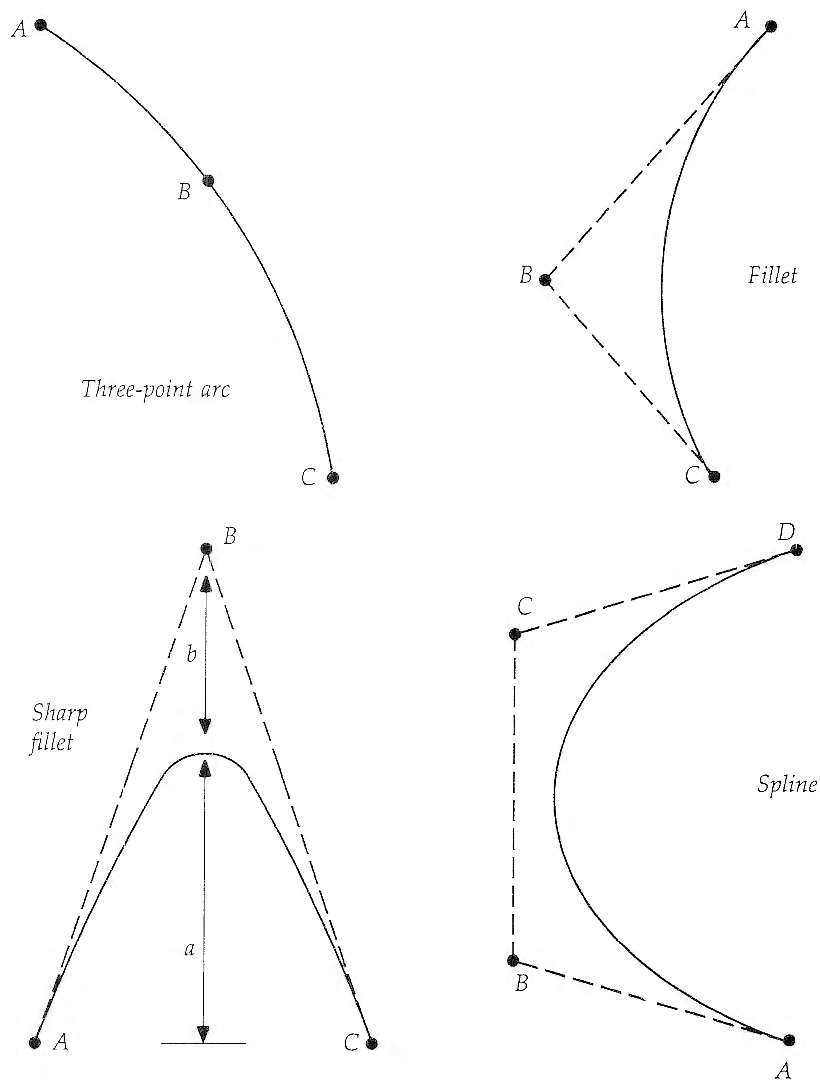


Figure 10-9. Other curves

another where they join. GpiPolyFillet can draw many fillets at once, joined to each other to produce complicated curves.

A variation of the fillet, the *sharp fillet,* allows the sharpness of the curve to be controlled by a separate argument to the function GpiPolyFilletSharp. This argument is called the *sharpness value.* In the figure this value is the ratio of *a/b.* When *a/b* is less than 1, the fillet is an ellipse, when it is 1, the fillet is a parabola, and when it is greater than 1, it's a hyperbola. GpiPolyFilletSharp can draw any number of sharp fillets.

The *spline* uses three control lines instead of two and is thus suitable for curves of a greater angle and complexity. It's created with GpiPolySpline, which can create many splines at once.

MARKER PRIMITIVES

Markers are small graphics objects such as crosses, diamonds, and stars. They are typically used to mark the data points on a line graph, but they have other purposes as well. PM makes available a dozen standard markers.

In this section we introduce markers, and also the concepts of foreground and background colors. We'll also examine a powerful new API: WinSetAttrs, which permits many attributes to be set at once.

Markers share some of the characteristics of text characters. In fact, custom markers can be defined using the same techniques as are used to create text fonts.

Our example program draws a marker on the screen wherever you click the mouse and connects the markers with lines. This creates a line graph, as shown in Figure 10-10.

Here are the listings for MARKER.C, MARKER.H, MARKER, and MARKER.DEF.

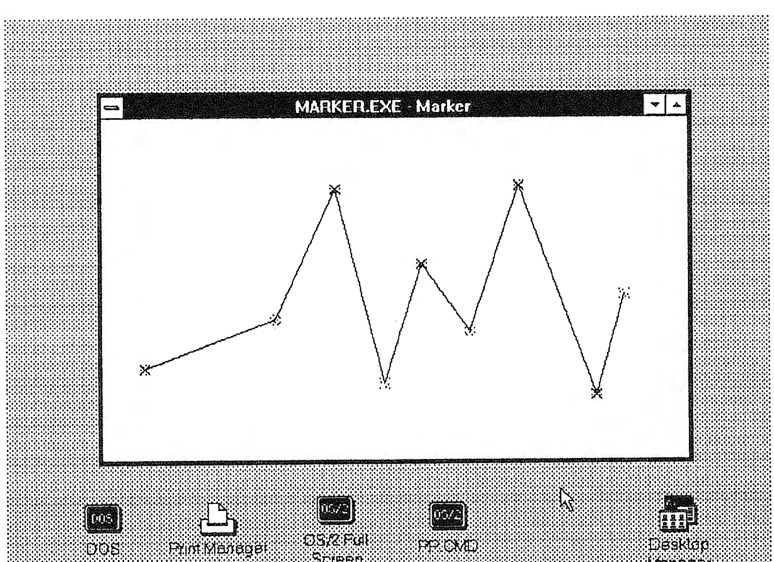


Figure 10-10. Output of the MARKER program

```
/* ----------------- */
/* MARKER - A marker */
/* ----------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "marker.h"

USHORT cdecl main(void)
   {
   HAB hab;                             /* handle for anchor block */
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd, hwndClient;               /* handles for windows */

                                        /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);     /* create message queue */

                                        /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                        /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Marker", 0L, NULL, 0, &hwndClient);

                                        /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);              /* destroy frame window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */

   return 0;
   }

                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   HPS hps;
   static POINTL ptl[MAXPTLS];
   static int i = 0;
   static MARKERBUNDLE mbnd = {CLR_RED, CLR_YELLOW, 0, BM_OVERPAINT,
                            MARKSYM_EIGHTPOINTSTAR, 0};

   switch (msg)
      {
      case WM_BUTTON1DOWN:
         if (i <= MAXPTLS)
            {                           /* add new marker */
            ptl[i].x = SHORT1FROMMP(mp1);
            ptl[i++].y = SHORT2FROMMP(mp1);
            WinInvalidateRect(hwnd, NULL, FALSE);
            }
```

```
         else                           /* too many markers */
            WinAlarm(HWND_DESKTOP, WA_WARNING);
         WinDefWindowProc(hwnd, msg, mp1, mp2);
         return TRUE;
         break;

      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
         GpiSetAttrs(hps, PRIM_MARKER, MBB_COLOR | MBB_BACK_COLOR |
            MBB_BACK_MIX_MODE, 0L, &mbnd);        /* set attrs */
         GpiPolyMarker(hps, (LONG)i, &ptl[0]);     /* draw markers */
         GpiMove(hps, &ptl[0]);
         GpiPolyLine(hps, (LONG)i - 1, &ptl[1]);   /* lines */
         WinEndPaint(hps);
         break;

      case WM_ERASEBACKGROUND:
         return TRUE;                   /* erase background */
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                         /* NULL / FALSE */
   }
```

```
/* -------- */
/* MARKER.H */
/* -------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define MAXPTLS 50
```

```
# -----------------
# MARKER Make file
# -----------------

marker.obj: marker.c marker.h
   cl -c -G2s -W3 -Zp marker.c

marker.exe: marker.obj marker.def
   link /NOD marker,,NUL,os2 slibce,marker
```

```
;-----------
; MARKER.DEF
; ----------

NAME         MARKER WINDOWAPI

DESCRIPTION 'Marker'

PROTMODE
STACKSIZE   4096
```

Each time the client window procedure receives a WM_BUTTON1-DOWN message, indicating that the user has clicked the mouse button, it places a point of type POINTL, representing the location of the mouse click, in the array *ptl*. It also invalidates the window so the WM_PAINT message will be received and the entire array of points and markers will be redrawn.

When a WM_PAINT message is received, the program obtains a cached micro PS with WinBeginPaint. It then sets various attributes of the markers using the GpiSetAttrs function; we'll explore this later. Finally the markers are placed at their proper positions with GpiPolyMarker, and lines are drawn to connect them using GpiPolyLine.

The GpiPolyMarker Function

A single marker can be placed with the GpiMarker function, but GpiPolyMarker allows a series of markers to be displayed using one function call. This function uses the same array of data points as GpiPolyLine, so once the array of points has been constructed, two function calls are all that's necessary to place all the markers and connect them with lines.

Draw a Series of Markers

```
LONG GpiPolyMarker(hps, cptl, aptl)
HPS hps            Handle to presentation space
LONG cptl          Number of points
PPOINTL aptl       Array of point structures

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

The default marker symbol is a cross, but a dozen other marker symbols are predefined and can be set using the GpiSetMarker function.

Specify Marker Symbol

```
BOOL GpiSetMarker(hps, lSymbol)
HPS hps            Handle to presentation space
LONG lSymbol       Marker symbol

Returns: GPI_OK if successful, GPI_ERROR if error
```

The identifier in the second argument specifies a marker in the default marker set, which can be one of the following:

Identifier	**Marker Symbol**
MARKSYM_CROSS	Thin X shape
MARKSYM_PLUS	Thin plus sign
MARKSYM_DIAMOND	Open diamond
MARKSYM_SQUARE	Open square
MARKSYM_SIXPOINTSTAR	Six-pointed star
MARKSYM_EIGHTPOINTSTAR	Eight-pointed star
MARKSYM_SOLIDDIAMOND	Solid diamond
MARKSYM_SOLIDSQUARE	Solid square
MARKSYM_DOT	Dot
MARKSYM_SMALLCIRCLE	Small circle
MARKSYM_BLANK	Nothing drawn

In our example we use GpiSetMarker to set the eight-pointed star. Different markers are typically used to distinguish different polylines on the same graph.

Foreground and Background Colors

Each marker is actually drawn inside a small box. This box forms the *background* of the marker. Both the marker itself and its background can be colored separately. This is a useful feature when, for example, the marker is the same color as the window it's drawn on. In this case coloring the marker's background in a contrasting color will make the marker visible. In this respect markers are like characters, which also have a foreground and background color. A marker with its background is shown in Figure 10-11.

The GpiSetAttrs Function

To set the marker colors, we use a function that can set the attributes of one primitive without affecting other primitives. This function, GpiSetAttrs, can also set several attributes at once. For instance, the color of the line primitive can be set to blue, without changing the color of other primitives; and the color and line type for a line primitive can be set simultaneously. The GpiSetAttrs function is versatile, but more complex than special-purpose functions like GpiSetColor and GpiSetLineType.

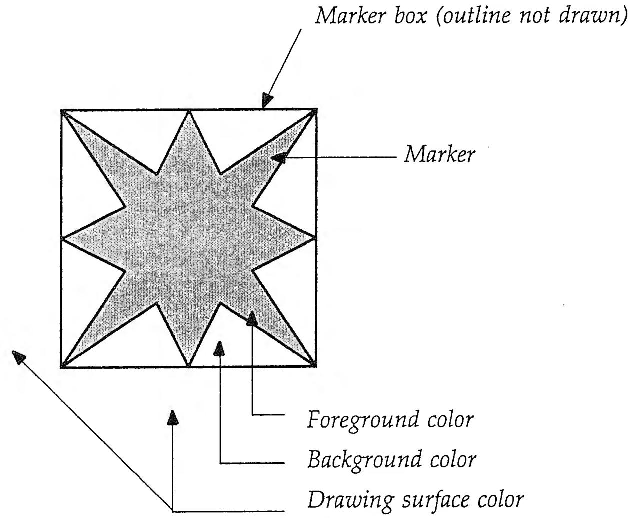


Figure 10-11. Marker foreground and background

Set Attributes for Specified Primitive

```
BOOL GpiSetAttrs(hps, flPrimType, flAttrsMask, flDefsMask,
                 pbunAttrs)
HPS hps              Handle to presentation space
LONG flPrimType      Primitive (PRIM_LINE, PRIM_AREA, etc.)
ULONG flAttrsMask    Attributes to set (LBB_COLOR, LBB_TYPE, etc.)
ULONG flDefsMask     Attributes to set to default
PBUNFLE pbunAttrs    Pointer to attribute structure

Returns: GPI_OK if successful, GPI_ERROR if error
```

The second argument, after the PS handle, is the primitive whose attribute you want to change. The possible values are

Identifier	**Primitive**
PRIM_LINE	Line and arc primitives
PRIM_MARKER	Marker primitive
PRIM_AREA	Area primitive
PRIM_CHAR	Character primitive
PRIM_IMAGE	Image primitive

In our MARKER example we use PRIM_MARKER for *flPrimType*.

The next two arguments are masks in which each bit position represents a particular attribute. Each primitive has its own set of attributes. Those for the marker attribute are

Identifier	Attribute
MBB_COLOR	Marker color
MBB_BACK_COLOR	Marker background color
MBB_MIX_MODE	Marker mix mode
MBB_BACK_MIX_MODE	Marker background mix mode
MBB_SET	Local font identifier
MBB_SYMBOL	Code point in font
MBB_BOX	Marker box size

The *flAttrsMask* argument contains the identifiers, ORed together, for those attributes we want to change. In our example there are three: MBB_COLOR, MBB_BACK_COLOR, and MBB_BACK_MIX_MODE. Attributes not specified here will not be changed.

The *flDefsMask* argument uses the same identifiers as the preceding argument. An identifier here indicates that we want to set an attribute to its default value. We don't want to set any of the attributes to their default value in this example, so we set this argument to 0L. Setting attributes to their default is useful after you've set them to something else, and want to change back to the original value.

The values to be given the attributes designated in *flAttrsMask* are stored in an array whose name ends in BUNDLE. We're changing the attributes of markers, so the array is type MARKERBUNDLE, which is defined like this in PMGPI.H:

```
typedef struct _MARKERBUNDLE /* mbnd */
   {
   LONG lColor;              /* marker color */
   LONG lBackColor;          /* background color */
   USHORT usMixMode;         /* marker mix mode */
   USHORT usBackMixMode;     /* background mix mode */
   USHORT usSet;             /* local font identifier */
   USHORT usSymbol;          /* code point in font */
   SIZEF sizfxCell;          /* marker box dimensions */
   } MARKERBUNDLE;
```

The *usSet* and *usSymbol* members are used only with markers taken from custom fonts; they don't apply here. The *sizfxCell* member can be used to change the size of boxes holding vector markers, but since the standard markers are bit images rather than vectors, this variable does not apply either.

Setting Colors

In the current example we initialize the marker foreground and background color values in the MARKERBUNDLE structure. These are the values that will be set when we call GpiSetAttrs. The foreground color (the marker itself) is initialized to CLR_RED, and the marker background is initialized to CLR_YELLOW.

As we saw earlier, foreground colors can be set with GpiSetColor instead of GpiSetAttrs. Similarly, background colors can be set with GpiSetBackColor. Remember that these functions set colors for *all* primitives, while GpiSetAttrs sets them only for a specified primitive. Foreground and background colors can be retrieved with GpiQueryColor and GpiQueryBackColor, while the attributes of a specific primitive can be retrieved with GpiQueryAttrs.

Setting Mix Modes

Mix modes specify how a graphics object of one color will combine with a drawing surface of another color. In the MARKERBUNDLE structure the foreground and background mix modes are specified by *usMixMode* and *usBackMixMode.* The default mix mode for marker foreground colors is FM_OVERPAINT, which means that the foreground color is simply drawn over the drawing surface color. This is what we want in our example, so we don't need to change the foreground mix mode.

The default background mix mode is BM_LEAVEALONE. This means that the background of our marker will never be seen: the drawing surface color will be "left alone." But we want the marker background color to be visible, so we change *usBackMixMode* to BM_OVERPAINT.

There are 16 foreground and background mix modes. We'll explore them in the next chapter when we talk about areas.

PRINTER OUTPUT

We've seen examples of PM graphics displayed on the screen. In this section we'll find out how to output these same graphics pictures to a printer.

PM has a major advantage when it comes to printing: device independence. In MS-DOS, an application needs to include a printer driver for any printer the application is likely to use, and there may be dozens. Also, to print graphics, the application must tailor its output to the specific printer.

It will send different output to a laser printer than to a dot matrix printer, for example. Both of these will be different from screen output.

In PM, the operating system itself supplies whatever printer drivers are necessary. They may be part of the original system, or installed later by the user. The application doesn't need to include any printer drivers. It ordinarily outputs graphics in a device-independent format, and the system takes care of seeing that the output is printed properly on whatever printer is connected to the system. The user can switch from one printer to another using the Print Manager; this change will be invisible to the application.

Another advantage of using PM for printing is that printer jobs can be automatically *spooled* or placed in a queue where they will await their turn for printing. Once the output is written to the queue, the application can go about other business. The system will take care of the printing when the printer is free.

There is a price to pay for these advantages. Printing itself is fairly easy, but the task of obtaining information about the printer, which is a necessary prerequisite to printing, is somewhat baroque.

Note that in PM, text is simply another form of graphics. While our example demonstrates graphics output, it applies equally well to text.

Overview of Printing

The philosophy in PM is to use the same graphics statements to draw on any device. If a series of graphics calls like GpiLine and GpiBox draws a picture on the screen, then this same series of Gpi calls will draw the same picture on the printer. The difference is that the device context for the screen is replaced by the device context for the printer.

As we noted at the beginning of the chapter, not all presentation spaces work with all devices. So far we've used the cached micro PS, which applies only to the screen. To output to the printer, we need either the non-cached micro PS or the normal PS. Since we're not using retain mode, the micro PS is all we need. While the cached micro PS is automatically associated with a device context for the screen, the program must explicitly open a device context for a micro PS. The micro PS can work with a variety of devices, although once associated with a particular device, it cannot be reassociated with another.

To open a device context for a micro PS, we use the DevOpenDC function. Here's one common approach to printing:

1. Open a device context for the printer using DevOpenDC.

2. Create a micro presentation space using GpiCreatePS and associate it with the device context.

3. Execute the Gpi functions to draw the picture.

4. Destroy the presentation space with GpiDestroyPS.

5. Close the device context with DevCloseDC.

The last four steps are fairly straightforward, but the first one requires considerable preparation, as we'll see.

The PRTGRAPH Example

Our example is derived from the MARKER program. For the user, the only change is that a menu item, "Print," has been added to the action bar. The user draws a graph using the mouse. Printing the graph requires only that the "Print" menu item be selected.

While in many programs the picture sent to the printer will be exactly the same as that sent to the screen, this doesn't need to be the case. Output to the printer is not in the form of a screen dump, but in a re-creation of the original picture. During the re-creation, the picture can be altered as desired. To demonstrate this flexibility, we don't print the markers. The polylines are the same in both cases. The printer output is shown in Figure 10-12.

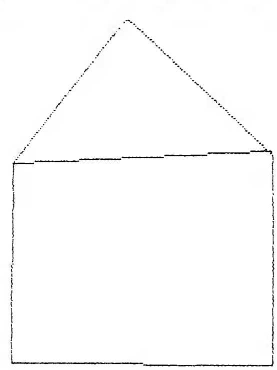


Figure 10-12. Printer output of the PRTGRAPH program

You should be aware that in some prerelease versions of OS/2 this program will only work if you disable the spooler with the Print Manager.

Here are the listings for PRTGRAPH.C, PRTGRAPH.H, PRTGRAPH, PRTGRAPH.DEF, and PRTGRAPH.RC.

```
/* ---------------------- */
/* PRTGRAPH - graph print */
/* ---------------------- */

#define INCL_GPI
#define INCL_WIN
#define INCL_DEV
#include <os2.h>
#include <string.h>

#include "prtgraph.h"

HAB hab;                              /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                           /* handle for message queue */
   QMSG qmsg;                         /* message queue element */
   HWND hwnd, hwndClient;             /* handles for windows */

                                      /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                      FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                      FCF_MENU;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);         /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);   /* create message queue */

                                      /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                      /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Prtgraph", 0L, NULL, ID_FRAMERC,
             &hwndClient);

                                      /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);            /* destroy frame window */
   WinDestroyMsgQueue(hmq);           /* destroy message queue */
   WinTerminate(hab);                 /* terminate PM usage */

   return 0;
   }

                                      /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                     MPARAM mp2)
   {
   HPS hps, hpsPrt;
   static SIZEL sizl = {0, 0};
```

```
static POINTL ptl[MAXPTLS];
static int i = 0;
static MARKERBUNDLE mbnd = {CLR_RED, CLR_YELLOW, 0, BM_OVERPAINT,
                           MARKSYM_EIGHTPOINTSTAR, 0,
                           MAKEFIXED (0, 0), MAKEFIXED(0, 0)};
DEVOPENSTRUC dop;
CHAR szPrinter[32];
CHAR szDetails[256];
USHORT cb;
HDC hdc;

switch (msg)
   {
   case WM_COMMAND:
      switch (COMMANDMSG(&msg) -> cmd)
         {
         case ID_PRINT:
            {
                                    /* get printer name */
            cb = WinQueryProfileString(hab, "PM_SPOOLER", "PRINTER", "",
               szPrinter, sizeof(szPrinter));
            szPrinter[cb - 2] = 0;  /* remove ";" */

            cb = WinQueryProfileString(hab, "PM_SPOOLER_PRINTER",
               szPrinter, "", szDetails, sizeof(szDetails));
            strtok(szDetails, ";");
            dop.pszDriverName = strtok(NULL, ";");
            dop.pszLogAddress = strtok(NULL, ";");
            strtok(dop.pszDriverName, ",");
            dop.pdriv = NULL;
            dop.pszDataType = "PM_Q_STD";
            hdc = DevOpenDC(hab, OD_QUEUED, "*", 4L, (PDEVOPENDATA)&dop,
                    (HDC)NULL);

            hpsPrt = GpiCreatePS(hab, hdc, &sizl, PU_LOMETRIC |
                        GPIT_MICRO | GPIA_ASSOC);

            GpiMove(hpsPrt, &ptl[0]);
            GpiPolyLine(hpsPrt, (LONG)i - 1, &ptl[1]);   /* lines */
            GpiDestroyPS(hpsPrt);
            DevCloseDC(hdc);
            break;
            }
         }
         break;

   case WM_BUTTON1DOWN:
      if (i <= MAXPTLS)
         {                          /* add new marker */
         ptl[i].x = SHORT1FROMMP(mp1);
         ptl[i].y = SHORT2FROMMP(mp1);
         hps = WinGetPS(hwnd);
         GpiSetPS(hps, &sizl, PU_LOMETRIC);
         GpiConvert(hps, CVTC_DEVICE, CVTC_PAGE, 1L, &ptl[i++]);
         WinReleasePS(hps);
         WinInvalidateRect(hwnd, NULL, FALSE);
         }
      else
         WinAlarm(HWND_DESKTOP, WA_WARNING);   /* too many markers */
      WinDefWindowProc(hwnd, msg, mp1, mp2);
      return TRUE;
```

```
         break;

      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
         GpiSetPS(hps, &sizl, PU_LOMETRIC);
         GpiSetAttrs(hps, PRIM_MARKER, MBB_COLOR ¦ MBB_BACK_COLOR ¦
            MBB_SYMBOL ¦ MBB_BACK_MIX_MODE, 0L, &mbnd);
         GpiPolyMarker(hps, (LONG)i, &ptl[0]);     /* markers */
         GpiMove(hps, &ptl[0]);
         GpiPolyLine(hps, (LONG)i - 1, &ptl[1]);   /* lines */
         WinEndPaint(hps);
         break;

      case WM_ERASEBACKGROUND:
         return TRUE;                        /* erase background */
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                              /* NULL / FALSE */
   }
```

```
/* ---------- */
/* PRTGRAPH.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define MAXPTLS 50

#define ID_FRAMERC 1
#define ID_PRINT 100
```

```
# ------------------
# PRTGRAPH Make file
# ------------------

prtgraph.obj: prtgraph.c prtgraph.h
   cl -c -G2s -W3 -Zp prtgraph.c

prtgraph.res: prtgraph.rc prtgraph.h
   rc -r prtgraph.rc

prtgraph.exe: prtgraph.obj prtgraph.def
   link /NOD prtgraph,,NUL,os2 slibce,prtgraph
   rc prtgraph.res

prtgraph.exe: prtgraph.res
   rc prtgraph.res
```

```
; ------------
; PRTGRAPH.DEF
; ------------

NAME       PRTGRAPH WINDOWAPI

DESCRIPTION 'Prtgraph'

PROTMODE
STACKSIZE   4096
```

```
/* ----------- */
/* PRTGRAPH.RC */
/* ----------- */

#include <os2.h>
#include "prtgraph.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "~Print", ID_PRINT
   END
```

As you can see in PRTGRAPH.C, printing takes place when a WM_COMMAND message with a command value of ID_PRINT is received. Let's analyze what happens in this section of the program.

Opening a Device Context for the Printer

The DevOpenDC function requires printer information that is stored in a special file. Extracting this information requires several steps.

The OS2.INI File

OS/2 keeps a file, called OS2.INI, in which it records (unfortunately, not in ASCII) information about the system configuration. This information consists of *profiles.* There is a profile for various peripherals, including the *default printer.* The default printer is the one that will normally be used for printing, as in this example. If there is more than one printer in the system, the user can specify the default from the Print Manager.

To open a device context for printing, DevOpenDC needs two pieces of information about the default printer: its *driver name* and its *logical device address.* The driver name is the file name of the printer driver, such as "IBM4201". The logical device address is the imaginary port currently used by the queue for the printer, such as "LPT1Q". It is sometimes called the

queue port. These strings are stored in OS2.INI. To retrieve them, we first need to find the name of the default printer itself, which is also stored in OS2.INI. (We said it was a baroque process.)

Information in OS2.INI is arranged first by *section* and then within each section by *keyname.* A section relates to a particular application, such as the Print Manager. The keyname is the heading for the specific piece of information. The information in each keyname is stored in the form of a text string.

The WinQueryProfileString Function

The API that extracts information from OS2.INI is WinQueryProfileString.

Retrieve String from OS2.INI File

```
USHORT WinQueryProfileString(hab, pszAppName, pszKeyName,
                             pszError, pszBuf, cchBuf)
HAB hab             Handle to anchor block
PSZ pszAppName      Application name
PSZ pszKeyName      Keyname
PSZ pszError        Default string (used if key not found)
PSZ pszBuf          Buffer for string
USHORT cchBuf       Size of buffer

Returns: Number of characters placed in pszBuf
```

We supply the *pszAppName* (or section name) and *pszKeyName.* The application then returns with the string we want in *pszBuf,* whose size we specify in *cchBuf.* We can optionally supply a string, *pszError,* that will be placed in *pszBuf* if the function can't find the section-and-keyname combination we supplied. The return value is the length of the string actually placed in *pszBuf.*

To find the name of the default printer, we use WinQueryProfileString to look in the section called "PM_SPOOLER" under the keyname "PRINTER". The name returned might be "PRINTER1".

We call WinQueryProfileString again to find the driver name and logical address, this time using a section name of "PM_SPOOLER_PRINTER" and the previously obtained printer name ("PRINTER1" or whatever it was) as the keyname.

Parsing the "Printer Details" String

If all goes well, WinQueryProfileString will return with a long string, *printer details,* in which four substrings are separated by semicolons. A typical printer details string might be

```
"LPT1;IBM4201;LPT1Q;;"
```

The substrings are the physical port, the driver name, the logical port (or queue port), and the network parameters (which in this example are not filled in).

The C library function *strtok,* which extracts substrings (tokens) from a string, is convenient for parsing the printer details string. The first call to *strtok* returns the physical port. We don't need this, so it goes into the bit bucket. The second call gets the driver name, which is one of the two substrings we do want. The second substring we want is the logical address, which we get with the third call to *strtok.*

Unfortunately, the parsing job is not quite complete. The driver name may consist of several names separated by commas. The default driver name is the first of these. To separate it from the others, we invoke *strtok* again, this time with a comma as the delimiter character. Finally we have the necessary information to call DevOpenDC.

The DevOpenDC Function

The DevOpenDC function creates a device context. We use it to open a device context for the printer. This function's prototype is in the PMDEV.H header file, so you need to *#define* INCL_DEV in your program.

Create Device Context

```
HDC DevOpenDC(hab, type, pszToken, count, pbData, hdcComp)
HAB hab                    Anchor block handle
LONG type                  Type of device context (OD_QUEUED, etc.)
PSZ pszToken               Device info token; if "*", not used
LONG count                 Number of items in pbData
PDEVOPENSTRUC pbData       Structure describing data
HDC hdcComp                Compatible device context (NULL is screen)

Returns: Device context handle if successful, DEV_ERROR if error
```

The first argument to this function is the anchor block handle obtained with WinInitialize. The second is the type of device context. There are several possibilities:

Identifier	Device Context Type
OD_QUEUED	Queued device (printer or plotter)
OD_DIRECT	Non-queued device (printer or plotter)
OD_INFO	Read information from device; no output
OD_METAFILE	Metafile
OD_MEMORY	Memory

In our example we use OD_QUEUED, since output to the printer is normally queued.

The device information token specified in the third argument is not used in the current version of OS/2. A string containing an asterisk indicates the string is not meaningful. The fourth argument is the count of the number of members in a structure of type DEVOPENSTRUC that will be used. We use the first four items in the structure. The fifth argument is a pointer to the structure. This structure is defined this way in OS2DEF.H:

```
typedef struct _DEVOPENSTRUC /* dop */
   {
   PSZ pszLogAddress;        /* Logical device address (lptl) */
   PSZ pszDriverName;        /* Device driver name (IBM4201) */
   PDRIVDATA pdriv;          /* DRIVDATA structure */
   PSZ pszDataType;          /* Device driver type (PM_Q_STD) */
   PSZ pszComment;           /* For queued devices */
   PSZ pszQueueProcName;     /* For queued devices */
   PSZ pszQueueProcParams;   /* For queued devices */
   PSZ pszSpoolerParams;     /* For queued devices */
   PSZ pszNetworkParams;     /* For queued devices */
   } DEVOPENSTRUC;
```

Into this structure we place the logical address and driver name we extracted so laboriously from OS2.INI. The *pdriv* member can be set to NULL, since we don't use the DRIVDATA structure to provide additional driver-specific information. The fourth argument, *pszDataType,* is the device driver type. We use "PM_Q_STD".

Creating the Presentation Space

Once the device context is open, the PS can be created with GpiCreatePS. Remember that this is not the same PS as that used to write to the screen. The PS handle here will be *hpsPrt,* while that for the screen is *hps.*

Create Presentation Space

```
HPS GpiCreatePS(hab, hdc, psizl, flOptions)
HAB hab              Anchor block handle
HDC hdc              Device context (needed if GPIA_ASSOC used)
PSIZEL psizl         Size of presentation page
ULONG flOptions      PS options (page unit, storage format, type)

Returns: Handle to presentation space if successful, GPI_ERROR if
         error
```

The first argument is the anchor block handle. The second specifies the device context with which this PS will be associated. In this case it's *hdcPrt,* returned from DevOpenDC. The third argument is the size of the presentation page, as specified by a structure of type SIZEL. If the size (0,0) is used, the presentation page will be the same size as the device page, which is usually what we want.

The fourth argument specifies the page units, and optionally the storage format, presentation type, and association option. The identifiers for the different options are ORed together.

The identifiers for the page units are the same as for GpiSetPS, shown earlier. They are

Identifier	**Page Unit Set To**
PU_ARBITRARY	Pixels initially; can be changed
PU_HIENGLISH	0.001 inch
PU_HIMETRIC	0.01 millimeter
PU_LOENGLISH	0.01 inch
PU_LOMETRIC	0.1 millimeter
PU_PELS	Pixels ("pels" to IBMers)
PU_TWIPS	Twip = a twentieth of a point (1/1440 inch)

In this example we use LO_METRIC, so units are 0.1 millimeter.

The storage format can be set to one of the following:

Identifier	**Storage Format for Coordinates**
GPIF_DEFAULT	Two-byte integers (default)
GPIF_LONG	Four-byte integers
GPIF_SHORT	Two-byte integers

We don't use a storage format option, so the default, two-byte integers, is used to represent coordinates.

The presentation type can be one of the following:

Identifier	Type of Presentation Space
GPIT_MICRO	Micro presentation space (use GPIA_ASSOC)
GPIT_NORMAL	Normal presentation space (default)

We're using a micro PS here, so the appropriate identifier is GPIT_MICRO.

The association type can be one of the following:

Identifier	Association Type
GPIA_ASSOC	Associates PS with *hdc* in second argument
GPIA_NOASSOC	Does not associate with device context

When a micro PS is specified, the GPIA_ASSOC type *must* be used (not the GpiAssociate function). This associates the PS with the previously obtained device context, used in the second argument. For a normal presentation space if GPIA_NOASSOC is used, the GpiAssociate function must be used and the second argument is ignored.

Drawing the Picture

Once the device context and presentation space are opened for the printer, we can create whatever picture we want by executing the appropriate Gpi functions. If the same APIs are used as those that create the picture on the screen, the same picture should appear on the printer. The Gpi functions that draw the picture could be placed in a function, and the function could be called both to draw to the screen and to draw to the printer. In our example, as we noted, we draw somewhat different pictures on the screen and on the printer.

Closing the PS and Device Context

When the drawing is finished, the presentation space is destroyed with GpiDestroyPS.

Destroy Presentation Space

```
BOOL GpiDestroyPS(hps)
HPS hps              Handle to presentation space

Returns: GPI_OK if successful, GPI_ERROR if error
```

Finally the device context is closed with DevCloseDC.

Close Device Context

```
HMF DevCloseDC(hdc)
HDC hdc             Handle to device context

Returns: DEV_OK (or metafile handle) if successful, DEV_ERROR if
error
```

This chapter has covered some of the simpler graphics primitives and attributes. In the next chapter we'll cover another graphics primative: fonts. In Chapter 12 we'll introduce areas, regions, paths, and bitmaps, and expand our discussion of color and mix modes.

11

FONTS AND TEXT

This chapter is concerned with displaying text. Since the Presentation Manager is completely graphics based, there is no restriction on the typeface or the size of text you can display. In a traditional character-based MS-DOS system, you can display characters in only one typeface and size. The characters are built into the hardware and are not normally changed. In PM the font is determined by software, not hardware. You can use 8-point Times Roman, 24-point Helvetica, and hundreds of other fonts in a wide range of sizes. Fonts can contain foreign-language characters, or special-purpose characters such as math symbols.

Several fonts come with PM (currently Courier, Helvetica, and Times Roman). Others are supplied by other vendors. You can even create your own fonts using a font editor. Fonts are stored in special DLL files with a .FON file extension.

The characters for any font will look the same on the printer as they do on the screen. PM is WYSIWYG—what you see is what you get.

We'll start this chapter by reviewing some terms. In the second section we'll show how to find what fonts are available in the system, load them, and display text with them. In the third section we'll explore the different

ways characters can be displayed. We'll finish up by showing how to control the position of each character in a line; the example will demonstrate text justification.

DEFINITIONS

So many specialized terms are used in printing that a discussion of fonts and text would be hard to understand for anyone not familiar with them. Some of these terms have been in use in the printing industry for hundreds of years; others are unique to the PM environment. This section by no means constitutes a complete discussion of printing terms—just those used to describe the operation of the example programs.

Characters

A *character* is a letter, number, punctuation mark, or other symbol as it is displayed or printed. 'A' and 'a' are characters, as are '7' and '#'. In PM, characters are considered a graphics primitive, like lines and areas.

Character Attributes

Like other primitives, characters can have attributes. These include foreground and background color and mix modes, and various aspects of the character's appearance in a line of text: its size, angle, shear, and direction.

Typefaces

A *typeface* is a set of characters of similar design. The name given to the typeface is the *face name* (or *typeface name*). Courier, Helvetica, and Times Roman are face names.

A typeface may be *proportional* or *fixed* width. In a proportional typeface the characters vary in width—an 'i' is narrower than an 'm'. Times Roman and Helvetica are proportional typefaces. In a fixed-width typeface all the characters are the same width, as they are on a typewriter. Courier is fixed width.

Fonts

A *font* is a collection of characters with the same typeface and height. The *height* is a general description of the size of the characters. Font heights are

traditionally measured in *points* (a unit used in printing since the 1700s). A point is 1/72 inch.

Don't confuse "font" with "face name." Two sets of characters with the same typeface, but different sizes, are different fonts. A font might be described as 12-point Times Roman. Ten-point Times Roman is a different font, since the size differs.

Font Characteristics

An existing font, like Helvetica 12 point, can be processed by PM to be *italic, bold, underscore,* or *strikeout.* These are called the font *characteristics.* PM slants characters to produce italics, makes character strokes wider to produce bold, draws a line under characters to produce underscore, and draws a line through the middle of characters to produce strikeout (which is used in legal documents).

To complicate the issue, a typeface may already have one of these characteristics designed in; you might load a typeface like Helvetica 12-point italic, for example. These are two different ways to make something italic. If you give the italic characteristic to Helvetica 12-point italic, it gets *more* italic.

Image and Outline Fonts

PM operates with fonts of two quite different kinds: *image fonts* and *outline fonts.* Image fonts are bitmaps: Each character is stored as a bitmap of the appropriate size. Outline fonts are vector fonts, created from lines and arcs. Outline fonts can be altered in a variety of ways, such as being scaled up or down in size, but their output quality is sometimes not as good as that of image fonts.

Public and Private Fonts

Fonts can be public or private. *Public fonts* are available to all applications running in the system. The user specifies which public fonts should be loaded by using the "Installation" menu in the Control Panel utility. The default system font is always available. Three other font families, Courier, Helvetica, and Times Roman, are supplied with the system and are available for installation through the Control Panel. Other fonts can be installed from diskettes. In the current version of OS/2 the default directory for public fonts is C:\OS2\DLL.

An application loads a *private font* from a DLL using the GpiLoadFonts function. The font is then available to the application, but not to the rest of the system.

The Font Editor

You can create your own image fonts using the font editor. However, creating a font requires considerable time and skill. We won't pursue this possibility here, nor will we discuss private fonts.

Font Metrics

PM stores a large amount of data about each font, including the font's dimensions. This data can be retrieved with the GpiQueryFonts or GpiQueryFontMetrics functions. Figure 11-1 shows some of the important font dimensions, which are called *font metrics.*

Each character is assumed to occupy a box called the *character cell.* The character cells are arranged along a *baseline.* The bottoms of most letters rest on the baseline; the exceptions are lowercase letters with descenders, like 'g' and 'p'. There is space in the character cell above the characters for accent marks. This is called *internal leading.* ("Leading" derives from the strips of lead old-time printers used to separate lines of text.) The font designer also suggests a certain spacing between the top of the character cells and the bottom of those on the line above. This is called *external leading.* The programmer can use the suggested value, or alter the line spacing by using different values. The distance from the bottom of the lowest character to the top of the tallest is the *maximum baseline extent.*

Logical and Physical Fonts

A *physical font* actually exists in the system, in the form of a DLL containing bitmap or vector definitions for the characters. When a line of text is printed, a physical font definition is used to form the letters.

What is a *logical font,* and why do we need it?

Suppose you create a document on your system using a particular font, say Helvetica 12 point. Later you send the document to another system which doesn't have this font. How can this second system print the document? The first system specifies a logical font that has the size, name, and other properties of Helvetica 12 point. If Helvetica 12 point is available on the second system, it will be used to print the document. If not, the system will find the closest available font and use that. The logical font is defined in terms of appearance, line weight, height, and other properties that an application wants to use to display text. It suggests, rather than demands, a particular physical font.

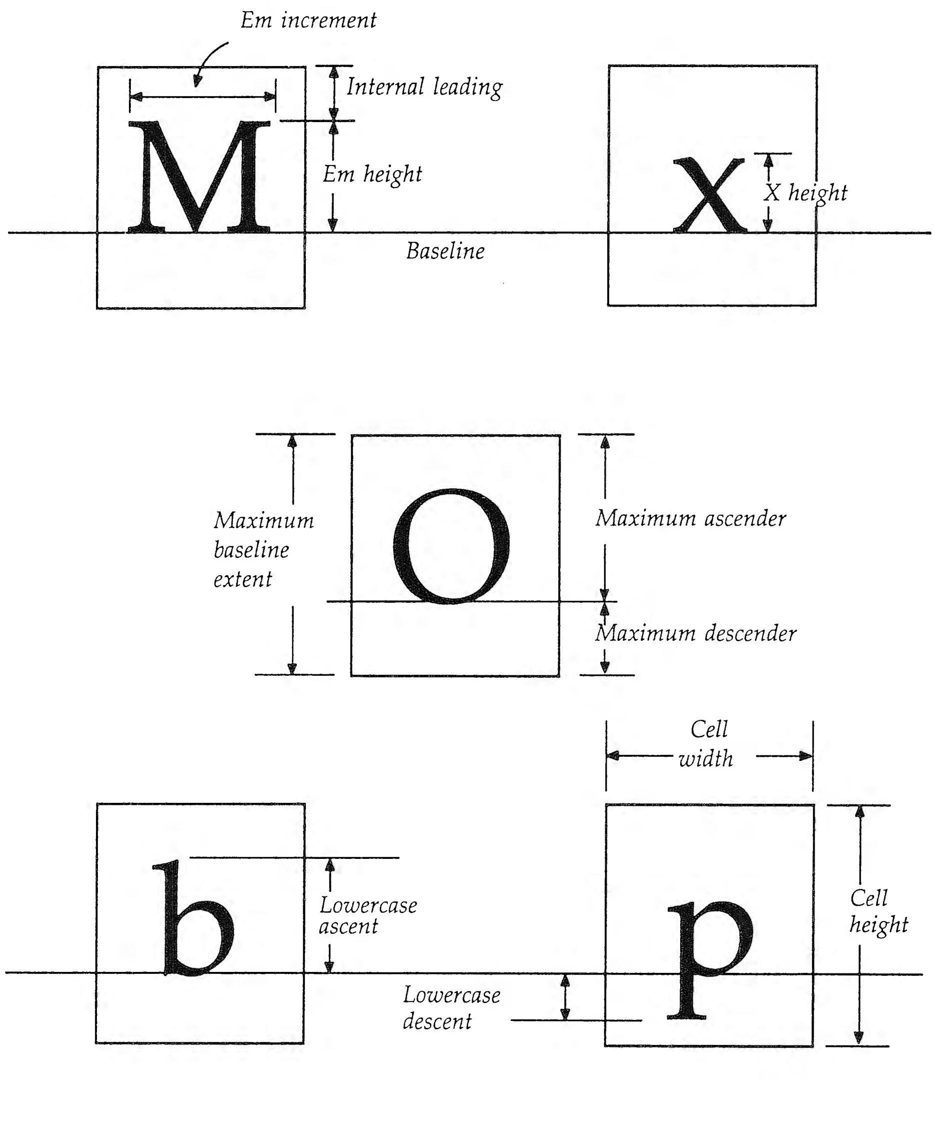


Figure 11-1. Font metrics

Physical and logical fonts are related in somewhat the same way as physical and logical color tables. One represents physical reality, the other represents an ideal that the system will try to match as closely as possible. The logical font approach permits great flexibility in creating and transferring documents.

FONTS

Our example program explores the properties of fonts by finding out what physical fonts are available in the system and then printing a line of text using each available font. When the screen is full, clicking on the "More" menu will fill it again.

Before you use this program, make sure that you have loaded the Courier, Helvetica, and Times Roman fonts from the appropriate .FON files, using the Control Panel.

Figure 11-2 shows some typical output for this program.

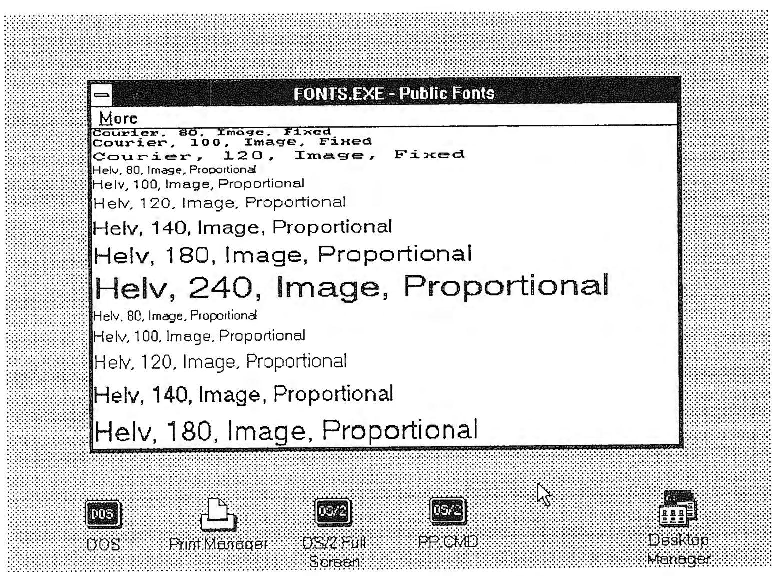


Figure 11-2. Output of the FONTS program

The font heights in the output are measured in decipoints. Thus 240 means 24 points, and so on.

Here are the listings for FONTS.C, FONTS.H, FONTS, and FONTS.DEF.

```
/* -------------------------------- */
/* FONTS - using the different fonts */
/* -------------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include <stdio.h>
#include <string.h>

#include "fonts.h"

USHORT cdecl main(void)
   {
   HAB hab;                              /* handle for anchor block */
   HMQ hmq;                              /* handle for message queue */
   QMSG qmsg;                            /* message queue element */
   HWND hwnd, hwndClient;                /* handles for windows */

                                         /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_MENU |
                         FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);            /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);      /* create message queue */

                                         /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                         /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Public Fonts", 0L, NULL, ID_FRAMERC,
             &hwndClient);

                                         /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);               /* destroy frame window */
   WinDestroyMsgQueue(hmq);              /* destroy message queue */
   WinTerminate(hab);                    /* terminate PM usage */

   return 0;
   }

                                         /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                     MPARAM mp2)
```

```
{
HPS hps;
static SHORT cy;
static SHORT frstFont = 0;
static FONTMETRICS fm[MAXFONTS];
static LONG cFonts = MAXFONTS;
static RECTL rcl = {0, 0, 0, 0};
SHORT top;
static FATTRS fat;
LONG lcBuf;
static int i;
CHAR szBuf[80];
CHAR *szType;
CHAR *szDefn;

switch (msg)
   {
   case WM_PAINT:
      hps = WinBeginPaint(hwnd, NULL, NULL);
      GpiErase(hps);

      top = 0;
      for (i = frstFont; i < cFonts; i++)
         {
         fat.lMatch = fm[i].lMatch;
         strcpy (fat.szFacename, fm[i].szFacename);

         GpiCreateLogFont(hps, NULL, 1L, &fat);   /* create font */

         GpiSetCharSet(hps, 1L);                  /* set new font */

         rcl.yTop = fm[i].lMaxBaselineExt + fm[i].lInternalLeading +
                    fm[i].lExternalLeading;
         if (top + rcl.yTop > cy) break;
         top += rcl.yTop;
         WinScrollWindow(hwnd, 0, rcl.yTop, NULL, NULL, NULL, NULL, 0);

         szType = (fm[i].fsType & 0x0001) ? "Fixed" : "Proportional";
         szDefn = (fm[i].fsDefn & 0x0001) ? "Vector" : "Image";
         lcBuf = sprintf(szBuf, "%s, %d, %s, %s", fm[i].szFacename,
                 fm[i].sNominalPointSize, szDefn, szType);
         WinDrawText(hps, lcBuf, szBuf, &rcl, 0, 0,
            DT_TEXTATTRS | DT_BOTTOM | DT_LEFT | DT_ERASERECT);

         GpiSetCharSet(hps, 0L);     /* release font */
         GpiDeleteSetId(hps, 1L);    /* delete font */
         }
      WinEndPaint(hps);
      break;

   case WM_COMMAND:
      switch (COMMANDMSG(&msg) -> cmd)
         {
         case ID_MORE: frstFont = i;   /* next fonts set */
         WinInvalidateRect(hwnd, NULL, FALSE);
         }
      break;

   case WM_SIZE:
      rcl.xRight = SHORT1FROMMP(mp2);
```

```
         cy = SHORT2FROMMP(mp2);
         break;

      case WM_CREATE:
         hps = WinGetPS(hwnd);
                                       /* query available public fonts */
         GpiQueryFonts(hps, QF_PUBLIC, NULL, &cFonts,
            sizeof(FONTMETRICS), &fm[0]);

         fat.usRecordLength = sizeof(fat);
         fat.fsSelection = 0;
         fat.idRegistry = 0;
         fat.usCodePage = 850;
         fat.lMaxBaselineExt = 0L;
         fat.lAveCharWidth = 0L;
         fat.fsType = 0;
         fat.fsFontUse = 0;
         WinReleasePS(hps);
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                        /* NULL / FALSE */
   }
```

```
/* ------- */
/* FONTS.H */
/* ------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_MORE 101

#define MAXFONTS 100
```

```
# ---------------
# FONTS Make file
# ---------------

fonts.obj: fonts.c fonts.h
   cl -c -G2s -W3 -Zp fonts.c

fonts.res: fonts.rc fonts.h
   rc -r fonts.rc

fonts.exe: fonts.obj fonts.def
   link /NOD fonts,,NUL,os2 slibce,fonts
   rc fonts.res

fonts.exe: fonts.res
   rc fonts.res
```

```
; ---------
; FONTS.DEF
; ---------

NAME        FONTS WINDOWAPI

DESCRIPTION 'Using different fonts'

PROTMODE
STACKSIZE   4096
```

During processing of the WM_CREATE message, the FONTS program first gathers information about all the physical fonts currently in the system and stores it in an array. Then, during processing of the WM_PAINT message, the program works its way through this array, using the information to set the current font to each of the available fonts in turn, and then displaying a line of text in that font.

Obtaining Font Information

During processing of the WM_CREATE message, our example program uses the GpiQueryFonts API to obtain information about all the available fonts in the system. This information is stored in a structure of type FONTMETRICS. (The program can handle up to MAXFONTS fonts, which is set to 100; you can change this if your system has more fonts.)

Retrieve Font Metrics for Loaded Fonts

```
LONG GpiQueryFonts(hps, flOptions, pszFacename, pcFonts, cbMetrics,
                   pfm)
HPS hps                Handle to presentation space
ULONG flOption         QF_PUBLIC or QF_PRIVATE
PSZ pszFacename        Typeface name (NULL for all fonts)
PLONG pcFonts          Number of fonts about which to receive data
LONG cbMetrics         Length of FONTMETRICS structure
PFONTMETRICS pfm       Array of FONTMETRICS structures

Returns: Number of fonts not returned if successful, GPI_ALTERROR
if error
```

The second parameter to GpiQueryFonts specifies whether the fonts queried will be public or private. We want public fonts, so we use QF_PUBLIC.

GpiQueryFonts can return information about a specific typeface, or it can return information about a list of typefaces installed in the system. In the first case the typeface name must be known in advance and is inserted in the third argument, *pszFacename*. In the example program GpiQueryFonts is used the second way: to obtain information about *all* public fonts in the system. Accordingly we set the third argument to NULL. The *cbMetrics* argument must be supplied with the size of the FONTMETRICS structure, and *pfm* is the pointer to the beginning of the array of FONTMETRICS (or to a single structure if only one font has been requested).

We're finally ready to examine the formidable FONTMETRICS structure:

```
typedef struct _FONTMETRICS  {  /* fm */
   CHAR szFamilyname[FACESIZE];  /* family name (Courier, etc.) */
   CHAR szFacename[FACESIZE];    /* typeface name (Courier, etc.) */
   USHORT idRegistry;         /* IBM registry number (0 if not registered) */
   USHORT usCodePage;         /* code page */
   LONG lEmHeight;            /* height of uppercase M */
   LONG lXHeight;             /* average lowercase character height */
   LONG lMaxAscender;         /* max height above baseline of any char */
   LONG lMaxDescender;        /* max depth below baseline of any char */
   LONG lLowerCaseAscent;     /* max height of any lowercase character */
   LONG lLowerCaseDescent;    /* max depth of any lowercase character */
   LONG lInternalLeading;     /* space above character for accent marks */
   LONG lExternalLeading;     /* space between rows of text */
   LONG lAveCharWidth;        /* average character width */
   LONG lMaxCharInc;          /* maximum increment between characters */
   LONG lEmInc;               /* width of uppercase M */
   LONG lMaxBaselineExt;      /* sum of max ascender and max descender */
   SHORT sCharSlope;          /* char slope angle (nonzero for italics) */
   SHORT sInlineDir;          /* rotation angle for text string */
   SHORT sCharRot;            /* baseline angle (part of font design) */
   USHORT usWeightClass;      /* thickness of character strokes */
   USHORT usWidthClass;       /* character width from 1 to 9 (5=normal) */
   SHORT sXDeviceRes;         /* horizontal resolution of target device */
   SHORT sYDeviceRes;         /* vertical resolution of target device */
   SHORT sFirstChar;          /* codepoint of first character in font */
   SHORT sLastChar;           /* codepoint of last character in font */
   SHORT sDefaultChar;        /* codepoint of default character */
   SHORT sBreakChar;          /* codepoint of space character */
   SHORT sNominalPointSize;   /* designed height of font in decipoints */
   SHORT sMinimumPointSize;   /* minimum height of font in decipoints */
   SHORT sMaximumPointSize;   /* maximum height of font in decipoints */
   USHORT fsType;             /* proportional or fixed, licensed, etc. */
   USHORT fsDefn;             /* 0=image font, 1=vector font */
   USHORT fsSelection;        /* italic, underscore, negative image, etc. */
```

```
   USHORT fsCapabilities;      /* 0=can be mixed with graphics, 1=not */
   LONG lSubscriptXSize;       /* horizontal size of subscripts */
   LONG lSubscriptYSize;       /* vertical size of subscripts */
   LONG lSubscriptXOffset;     /* horizontal offset from left of char cell */
   LONG lSubscriptYOffset;     /* vertical offset from baseline */
   LONG lSuperscriptXSize;     /* horizontal size of superscripts */
   LONG lSuperscriptYSize;     /* vertical size of superscripts */
   LONG lSuperscriptXOffset;   /* horiz offset from left edge of char cell */
   LONG lSuperscriptYOffset;   /* vert offset from left edge of char cell */
   LONG lUnderscoreSize;       /* width of underscore */
   LONG lUnderscorePosition;   /* distance from baseline to underscore */
   LONG lStrikeoutSize;        /* width of overstrike */
   LONG lStrikeoutPosition;    /* height of overstrike */
   SHORT sKerningPairs;        /* number of kerning pairs in table */
   SHORT sFamilyClass;         /* font family class and subclass */
   LONG lMatch;                /* copy to FATTRS with GpiCreateLogFont */
} FONTMETRICS;
```

This structure contains a great deal of information about a font, including dimensions for superscripts, subscripts, underscores, and overstrikes.

Creating a Logical Font

To change the current font, you need to do two things. The first is to create a logical font, and the second is to set the current font to this logical font.

The GpiCreateLogFont function is used to create the logical font.

Create a Logical Font

```
LONG GpiCreateLogFont(hps, pchName, lcid, pfat)
HPS hps             Handle to presentation space
PSTR8 pchName       Logical font name
LONG lcid           Local identifier
PFATTRS pfat        Structure of logical font properties

Returns: 2 if matching font found, 1 if not found, 0 if error
```

The second argument to this function is the logical font name, which is not used in this example. The third argument is given a value, made up by the application, that will be the *local identifier* for the font. This identifier is

used for all subsequent references to the font. It must be in the range from 1L to 254L.

The fourth argument is the address of a structure of type FATTRS. (This stands for Font Attributes, although the term "attributes" is not really appropriate here.) FATTRS describes the font we want. Since we're creating a logical font to access a physical font that we already know exists, we don't need to specify the font's properties in detail. Thus many of the parameters to this structure are given 0 values.

```
typedef struct _FATTRS  {       /* fat */
   USHORT usRecordLength;       /* length of this structure in bytes */
   USHORT fsSelection;          /* italic, bold, etc. */
   LONG lMatch;                 /* font match number (0=closest font) */
   CHAR szFacename[FACESIZE];   /* typeface name */
   USHORT idRegistry;           /* registry number */
   USHORT usCodePage;           /* code page identifier */
   LONG lMaxBaseLineExt;        /* sum max ascender and max descender */
   LONG lAveCharWidth;          /* average character width */
   USHORT fsType;               /* fixed or proportional, kerned or not */
   USHORT fsFontUse;            /* mix with graphics, outline, etc. */
   } FATTRS;
```

In our example the key parameter in this structure is *lMatch,* the match number. This is a unique number for a specific font. Its value was returned in FONTMETRICS, and by inserting it in FATTRS, we specify that we want this font and no other will do. This is called "forcing a match." (If we use 0 for this parameter, PM will find the closest physical font that matches the logical font whose properties we specify.)

As we noted, we set most of these parameters in this structure to 0, with three exceptions. The *usCodePage* argument is set to 850 for the multilingual code page. (We'll say more about code pages in the chapter on foreign language support.) The *szFacename* and *lMatch* arguments are set to the values obtained from the FONTMETRICS structure. These items are all GpiCreateLogFont needs to create a logical font that is based on an existing physical font.

Setting the Character Set

Once a logical font has been created, the current font can be set to this logical font with the GpiSetCharSet function. When this API is executed, all subsequent text will be displayed in the new font.

Set Character Set

```
BOOL GpiSetCharSet(hps, lcid)
HPS hps             Handle to presentation space
LONG lcid           Local identifier

Returns: GPI_OK if successful, GPI_ERROR if error
```

The second argument is given the same value that was used to identify the font in GpiCreateLogFont. (If this argument is set to 0L, the default character set is implied.)

Scrolling

The WinScrollWindow function is used to scroll the text upward when the "More" menu is selected. This function permits the contents of the window, or part of a window, to be moved up or down, or left or right.

Scroll Window Contents

```
SHORT WinScrollWindow(hwnd, dx, dy, prclScroll, prclClip, hrgnUpdate,
                      prclUpdate, fs)
HWND hwnd          Handle of window to be scrolled
SHORT dx           Amount of horizontal scrolling (device units)
SHORT dy           Amount of vertical scrolling (device units)
PRECTL prclScroll  Scroll rectangle (NULL=entire window)
PRECTL prclClip    Clip rectangle
HRGN hrgnUpdate    Handle of region to hold invalid region
PRECTL prclUpdate  Rectangle invalidated by scrolling
USHORT fs          SW_SCROLLCHILDREN, SW_INVALIDATERGN

Returns: Code indicating the type of invalid region
```

The second and third arguments specify the amount of horizontal and vertical scrolling. In the example, we don't scroll horizontally. The window

is scrolled vertically by the height of the text line, which is found by adding *lMaxBaselineExt* and *lExternalLeading* from FONTMETRICS.

The *prclScroll* argument can be used to scroll a smaller area than the entire window. We scroll the entire window, so this is set to NULL. The *prclClip* argument can define a clipping region different from that being scrolled. We don't use it, so it's set to NULL. The *hrgnUpdate* argument may be used to specify a region that will be set to the region being scrolled. This isn't needed, so it's set to NULL. The *prclUpdate* argument will be filled in with the coordinates of the rectangle invalidated by the scrolling operation. We don't need this information, so again this argument is set to NULL. Finally the *fs* argument can be SW_SCROLLCHILDREN to indicate that all child windows are scrolled along with the window being scrolled, or SW_INVALIDATERGN to indicate that the invalid region created by the scroll will be added to the window's update region. Neither of these is necessary here, so this argument is set to 0.

The return values from WinScrollWindow indicate the type of invalid region created by the scrolling operation. NULLREGION means there was no invalid region, SIMPLEREGION means a rectangular invalid region, and COMPLEXREGION means a nonrectangular region. ERROR means an error. To simplify the example we do not check the return code. However, this shortcut can create a visual problem if the window is overlaid by another window during scrolling. In a real application you should check the return code and repaint any invalidated region.

Displaying Text with WinDrawText

As we've done in previous chapters, we use WinDrawText to display the text. (In the next section we'll see that there are other, more versatile functions for displaying text.) The line of text includes various data items from FONTMETRICS that describe the font used to display the line. These are the font family name, the font face name, the nominal point size, and whether the font is vector or image, and fixed or proportional.

When you're through with a font, it should be deleted from the presentation space. Before this can be done, however, the particular character set must be disconnected from the current font. Thus after each font has been displayed, GpiSetCharSet is executed again with a local identifier of 0L to disconnect the current font. Then GpiDeleteSetId removes the local ID and deletes the logical font it represented.

MODIFYING AND DISPLAYING TEXT

Once a font is selected, displaying text using the font is simple. A function like WinDrawText or GpiCharStringAt displays a string at a particular location on the screen (or printer). What's more complicated, and interesting, are the changes you can make to the character box before displaying the text. Altering the character box can have a major effect on the appearance of the text.

Modifying the Character Box

You can change the character box in a variety of ways. Three of the most common are changes to its size, its angle, and its shear.

Figure 11-3 shows these modifications made to the character box.

The *box size* can be altered in both the X and Y directions, so the character can have any size or aspect ratio.

The *character angle* tilts the entire character box. A line drawn from the origin (the lower left corner of the window) through a specified point determines the angle. Any point on the line will do; thus (2,1) specifies the same angle as (10,5). The default (0 angle) is a horizontal line from the origin, specified by the point (1,0).

Shear changes the character box into a parallelogram. If you imagine the bottom of the box as being fixed, shear slides the top to the left or right. The angle of the shear is determined the same way as the direction angle: by specifying a point on a line drawn from the origin. The default (0 angle) is a vertical line from the origin, specified by the point (0,1).

Our example program, FANCYTXT, allows these three character box attributes—size, angle, and shear—to be altered, using menus. Also, the font characteristics—italic, underscore, strikeout, and bold—can be selected from a menu. Several of these characteristics can be in effect at once; these are indicated by check marks in the menu. Finally, different fonts can be selected. The selected font is also checked in the menu. (As in the previous example, make sure that Courier, Helvetica, and Times Roman have been loaded into your system.)

The phrase "Testing 123" is displayed, starting at the center of the screen. Figure 11-4 shows how this looks with the Times Roman typeface and no modification to the character box. Figure 11-5 shows the display when the character box has been expanded in the Y direction and the text has been underlined. Figure 11-6 shows the character angle changed and the text in Courier bold. Figure 11-7 shows the character box sheared to the left, with the Helvetica typeface.

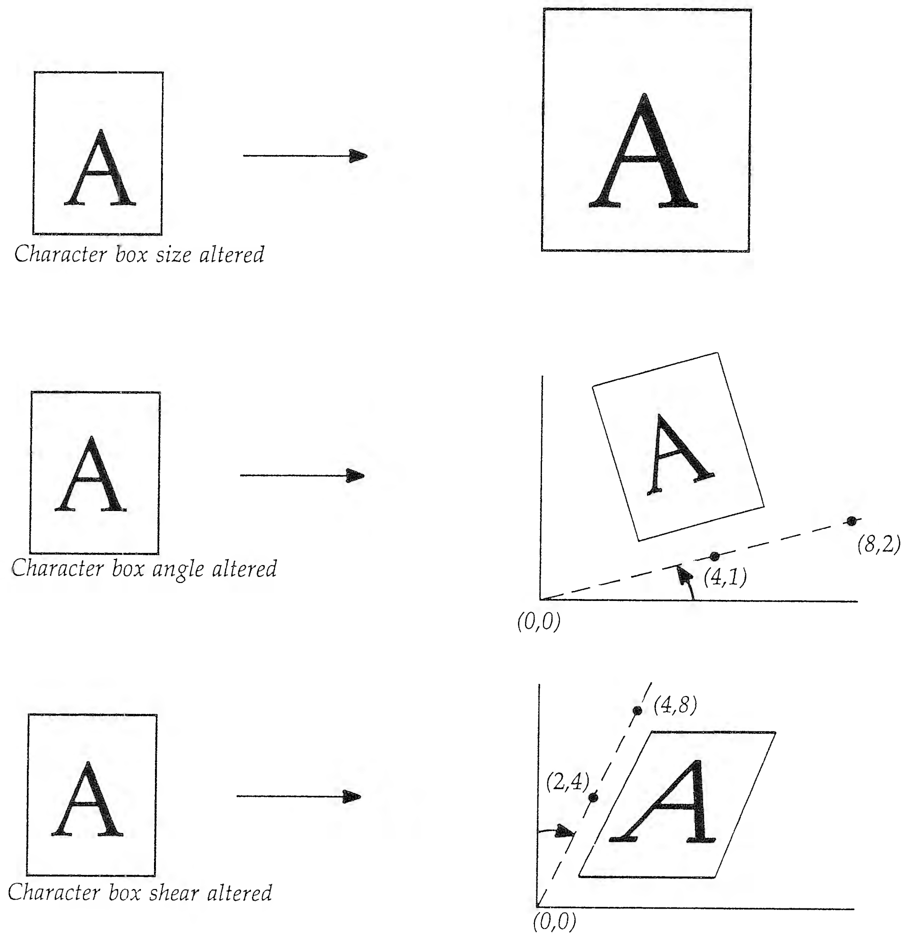


Figure 11-3. Character box, size, angle, and shear

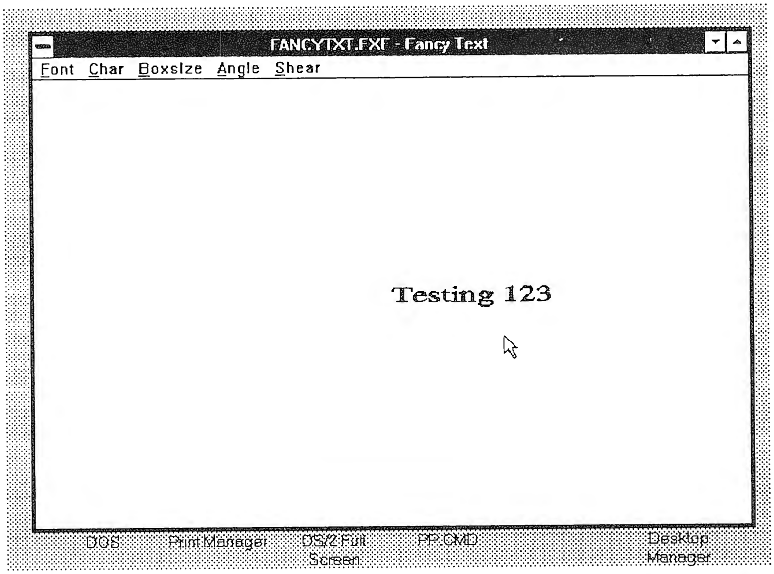


Figure 11-4. Normal character box

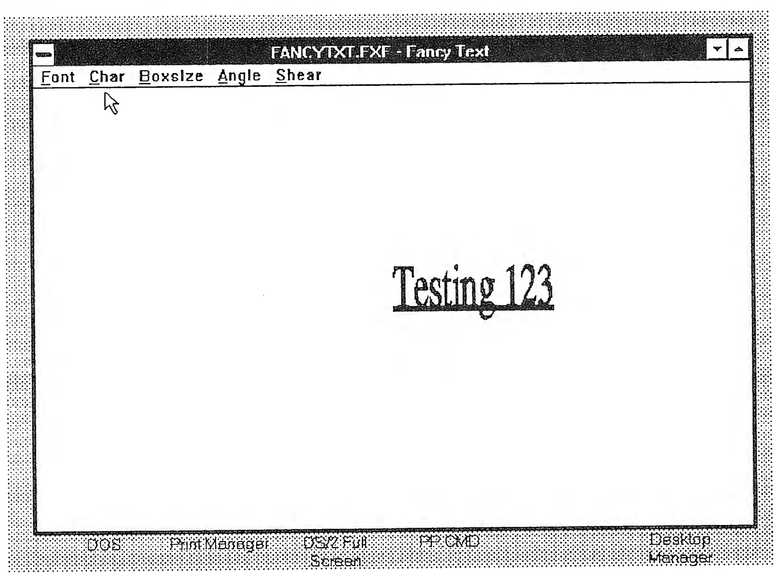


Figure 11-5. Character box made taller, text underscored

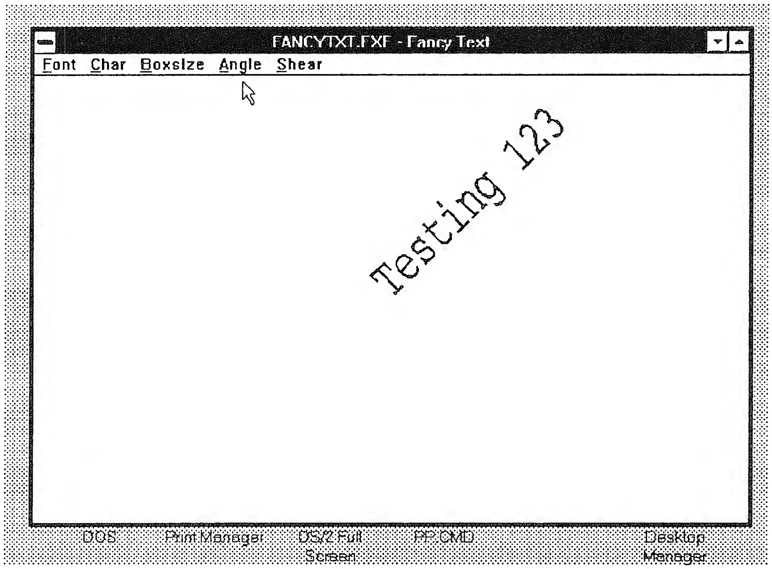


Figure 11-6. Tilted character box, text in bold

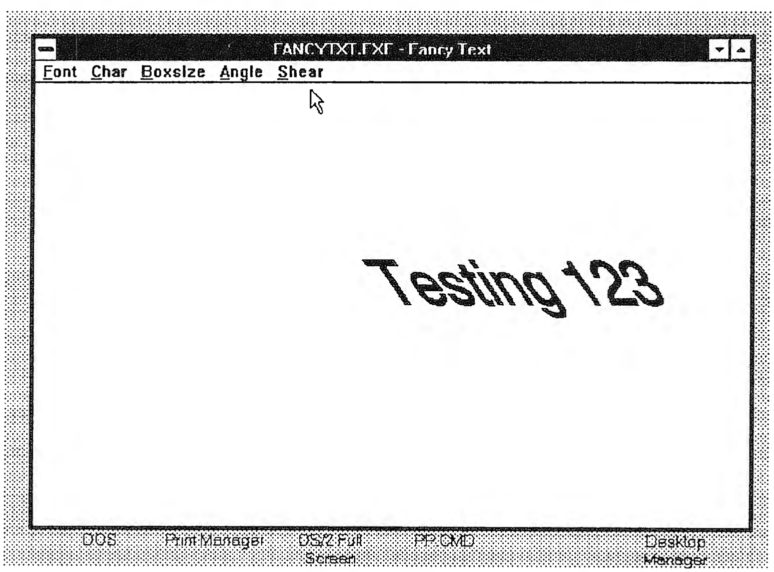


Figure 11-7. Sheared character box

Besides the alterations to the character box just noted, the order in which the character boxes are drawn can be reversed. Instead of text being printed left to right, as in English, it can be printed from right to left. This is called its *direction.* Changing the direction is useful for languages like Hebrew and Arabic, and for special effects. Our example program doesn't demonstrate this attribute.

You can also perform transformations on the character box. We'll discuss transformations in Chapter 13.

Character Modes

We mentioned earlier that there are image (raster) fonts and outline (vector) fonts. Image fonts can be used in two different ways, which results in three different *character modes.* The mode must be set with GpiSetCharMode before text can be printed.

- In *mode 1* the font is an image font that will not be resized or altered in any way, no matter what changes are made to the character box. You can't change the size, angle, or shear.
- In *mode 2* the font is again an image font, but it attempts to reflect some changes in the character box. If the character angle is changed, each character box will not be tilted, but the entire line of text will be, so the characters appear as if on a staircase. If the character box size is changed, the character size is unchanged, but the spacing between characters is. Changing the shear has no effect.
- In *mode 3* the font must be an outline font. All changes made to the character box will appear in the displayed text.

If you're using an image font, you would usually use mode 1, unless you wanted a special effect, in which case mode 2 might be appropriate. If you're using an outline font, you should use mode 3.

Here are the listings for FANCYTXT.C, FANCYTXT.H, FANCYTXT, FANCYTXT.DEF, and FANCYTXT.RC.

```
/* ---------------------------- */
/* FANCYTXT - Create fancy text */
/* ---------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>
```

```
#include <stdio.h>
#include <string.h>

#include "fancytxt.h"

HAB hab;                                /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd, hwndClient;               /* handles for windows */

                                        /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                      FCF_MINMAX | FCF_SHELLPOSITION | FCF_MENU |
                      FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);     /* create message queue */

                                        /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                        /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
            szClientClass, " - Fancy Text", 0L, NULL, ID_FRAMERC,
            &hwndClient);

                                        /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);              /* destroy frame window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */

   return 0;
   }

                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   static HDC hdc;
   static HPS hps;
   static HWND hwndMenu;                /* menu window handle */
   static SHORT idFont = ID_COURIERFONT;
   static SIZEL sizl = {0, 0};
   static POINTL ptlStart = {0, 0};
   static FATTRS fat;
   static SIZEF sizfxCharBox = {MAKEFIXED(500,0), MAKEFIXED(700,0)};
   static GRADIENTL  gradl = {1L, 0L};
   static POINTL ptlShear = {0L, 1L};
   USHORT sChecked;

   switch (msg)
      {
```

```
case WM_COMMAND:
   switch (COMMANDMSG(&msg) -> cmd)
      {                              /* facename change */
      case ID_COURIERFONT: strcpy (fat.szFacename, "Courier"); break;
      case ID_HELVFONT: strcpy (fat.szFacename, "Helv"); break;
      case ID_TMSRMNFONT: strcpy (fat.szFacename, "Tms Rmn"); break;

                                     /* characteristics change */
      case ID_ITALICATTR: fat.fsSelection ^= FATTR_SEL_ITALIC; break;
      case ID_UNDERSCOREATTR:
        fat.fsSelection ^= FATTR_SEL_UNDERSCORE;
        break;
      case ID_STRIKEOUTATTR: fat.fsSelection ^= FATTR_SEL_STRIKEOUT;
        break;
      case ID_BOLDATTR: fat.fsSelection ^= FATTR_SEL_BOLD; break;

                                     /* box size change */
      case ID_BOXINCX:
         sizfxCharBox.cx = MAKEFIXED(FIXEDINT(sizfxCharBox.cx) + 100,
                                     0);
         break;

      case ID_BOXDECX:
         sizfxCharBox.cx = MAKEFIXED(FIXEDINT(sizfxCharBox.cx) - 100,

                                     0);
         break;
      case ID_BOXINCY:
         sizfxCharBox.cy = MAKEFIXED(FIXEDINT(sizfxCharBox.cy) + 100,
                                     0);
         break;
      case ID_BOXDECY:
         sizfxCharBox.cy = MAKEFIXED(FIXEDINT(sizfxCharBox.cy) - 100,

                                     0);
         break;
                                     /* angle change */
      case ID_ANGLEINCX: gradl.x++; break;
      case ID_ANGLEDECX: gradl.x--; break;
      case ID_ANGLEINCY: gradl.y++; break;
      case ID_ANGLEDECY: gradl.y--; break;
                                     /* shear change */
      case ID_SHEARINCX: ptlShear.x++; break;
      case ID_SHEARDECX: ptlShear.x--; break;
      case ID_SHEARINCY: ptlShear.y++; break;
      case ID_SHEARDECY: ptlShear.y--; break;
      }

   switch (COMMANDMSG(&msg) -> cmd)
      {
      case ID_COURIERFONT:
      case ID_HELVFONT:
      case ID_TMSRMNFONT:
      case ID_ITALICATTR:
      case ID_UNDERSCOREATTR:
      case ID_STRIKEOUTATTR:
      case ID_BOLDATTR:
        GpiSetCharSet(hps, 0L);
        GpiDeleteSetId(hps, LCID);
        GpiCreateLogFont(hps, NULL, LCID, &fat);
        GpiSetCharSet(hps, LCID);
```

```
          if (COMMANDMSG(&msg) -> cmd < ID_CHARACTER)
            {                       /* font change */
                                    /* uncheck old */
           WinSendMsg(hwndMenu, MM_SETITEMATTR,
              MPFROM2SHORT(idFont, TRUE),
              MPFROM2SHORT(MIA_CHECKED, ~MIA_CHECKED));

           idFont = COMMANDMSG(&msg) -> cmd;

                                    /* check new */
           WinSendMsg(hwndMenu, MM_SETITEMATTR,
              MPFROM2SHORT(idFont, TRUE),
              MPFROM2SHORT(MIA_CHECKED, MIA_CHECKED));
            }
          else
            {                       /* attr. change */
                                    /* query checked? */

            sChecked = (USHORT) WinSendMsg(hwndMenu, MM_QUERYITEMATTR,
               MPFROM2SHORT(COMMANDMSG(&msg) -> cmd, TRUE),
               MPFROMSHORT(MIA_CHECKED));

                                    /* change checked state */
           WinSendMsg(hwndMenu, MM_SETITEMATTR,
              MPFROM2SHORT(COMMANDMSG(&msg) -> cmd, TRUE),
              MPFROM2SHORT(MIA_CHECKED, ~sChecked));
            }
          break;

        case ID_BOXINCX:
        case ID_BOXDECX:
        case ID_BOXINCY:
        case ID_BOXDECY:
          GpiSetCharBox(hps, &sizfxCharBox);
          break;

        case ID_ANGLEINCX:
        case ID_ANGLEDECX:
        case ID_ANGLEINCY:
        case ID_ANGLEDECY:
          GpiSetCharAngle(hps, &gradl);
          break;

        case ID_SHEARINCX:
        case ID_SHEARDECX:
        case ID_SHEARINCY:
        case ID_SHEARDECY:
           GpiSetCharShear(hps, &ptlShear);
           break;
        }

   WinInvalidateRect(hwnd, NULL, FALSE);
   break;

case WM_PAINT:
   hps = WinBeginPaint(hwnd, hps, NULL);
   GpiErase(hps);

   GpiCharStringAt(hps, &ptlStart, 11L, "Testing 123");

   WinEndPaint(hps);
   break;
```

```
      case WM_SIZE:
         ptlStart.x = SHORT1FROMMP(mp2) / 2;
         ptlStart.y = SHORT2FROMMP(mp2) / 2;
         GpiConvert(hps, CVTC_DEVICE, CVTC_WORLD, 1L, &ptlStart);
         break;

      case WM_CREATE:
                                          /* get menu window handle */
         hwndMenu = WinWindowFromID(WinQueryWindow(hwnd, QW_PARENT, FALSE),
                       FID_MENU);

         fat.usRecordLength = sizeof(fat);
         fat.fsSelection = 0;
         fat.lMatch = 0;
         strcpy (fat.szFacename, "Courier");
         fat.idRegistry = 0;
         fat.usCodePage = 850;
         fat.lMaxBaselineExt = 0L;
         fat.lAveCharWidth = 0L;
         fat.fsType = FATTR_TYPE_KERNING;
         fat.fsFontUse = FATTR_FONTUSE_OUTLINE;

         hdc = WinOpenWindowDC(hwnd);
         hps = GpiCreatePS(hab, hdc, &sizl, PU_TWIPS | GPIT_MICRO |
                  GPIA_ASSOC);

         GpiCreateLogFont(hps, NULL, LCID, &fat);
         GpiSetCharSet(hps, LCID);

         GpiSetCharMode(hps, CM_MODE3);    /* mode 3 */
         break;

      case WM_DESTROY:
         GpiSetCharSet(hps, 0L);
         GpiDeleteSetId(hps, LCID);
         GpiDestroyPS(hps);
                                          /* no close for window DC */
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                           /* NULL / FALSE */
   }
```

```
/* ---------- */
/* FANCYTXT.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define LCID 1L

#define ID_FRAMERC 1
```

```
#define ID_FONT 100
#define ID_COURIERFONT 101
#define ID_HELVFONT 102
#define ID_TMSRMNFONT 103

#define ID_CHARACTER 110
#define ID_ITALICATTR 111
#define ID_UNDERSCOREATTR 112
#define ID_STRIKEOUTATTR 113
#define ID_BOLDATTR 114

#define ID_BOXSIZE 120
#define ID_BOXINCX 121
#define ID_BOXDECX 122
#define ID_BOXINCY 123
#define ID_BOXDECY 124

#define ID_ANGLE 130
#define ID_ANGLEINCX 131
#define ID_ANGLEDECX 132
#define ID_ANGLEINCY 133
#define ID_ANGLEDECY 134

#define ID_SHEAR 140
#define ID_SHEARINCX 141
#define ID_SHEARDECX 142
#define ID_SHEARINCY 143
#define ID_SHEARDECY 144
```

```
# ------------------
# FANCYTXT Make file
# ------------------

fancytxt.obj: fancytxt.c fancytxt.h
   cl -c -G2s -W3 -Zp fancytxt.c

fancytxt.res: fancytxt.rc fancytxt.h
   rc -r fancytxt.rc

fancytxt.exe: fancytxt.obj fancytxt.def
   link /NOD fancytxt,,NUL,os2 slibce,fancytxt
   rc fancytxt.res

fancytxt.exe: fancytxt.res
   rc fancytxt.res
```

```
; ------------
; FANCYTXT.DEF
; ------------

NAME       FANCYTXT WINDOWAPI

DESCRIPTION 'Fancy text'

PROTMODE
STACKSIZE   4096
```

```
/* ----------- */
/* FANCYTXT.RC */
/* ----------- */

#include <os2.h>
#include "fancytxt.h"

MENU ID_FRAMERC
   BEGIN
      SUBMENU "~Font", ID_FONT
         BEGIN
            MENUITEM "~Courier", ID_COURIERFONT, , MIA_CHECKED
            MENUITEM "~Helv", ID_HELVFONT
            MENUITEM "~Tms Rmn", ID_TMSRMNFONT
         END

      SUBMENU "~Characteristics", ID_CHARACTER
         BEGIN
            MENUITEM "~Italic", ID_ITALICATTR
            MENUITEM "~Underscore", ID_UNDERSCOREATTR
            MENUITEM "~Strikeout", ID_STRIKEOUTATTR
            MENUITEM "~Bold", ID_BOLDATTR
         END

      SUBMENU "~Boxsize", ID_BOXSIZE
         BEGIN
            MENUITEM "X+100", ID_BOXINCX
            MENUITEM "X-100", ID_BOXDECX
            MENUITEM "Y+100", ID_BOXINCY
            MENUITEM "Y-100", ID_BOXDECY
         END

      SUBMENU "~Angle", ID_ANGLE
         BEGIN
            MENUITEM "X+1", ID_ANGLEINCX
            MENUITEM "X-1", ID_ANGLEDECX
            MENUITEM "Y+1", ID_ANGLEINCY
            MENUITEM "Y-1", ID_ANGLEDECY
         END

      SUBMENU "~Shear", ID_SHEAR
         BEGIN
            MENUITEM "X+1", ID_SHEARINCX
            MENUITEM "X-1", ID_SHEARDECX
            MENUITEM "Y+1", ID_SHEARINCY
            MENUITEM "Y-1", ID_SHEARDECY
         END
   END
```

This is a long listing—mostly because there are so many menu items to process—but the code to perform individual actions isn't hard to follow.

Let's start with the WM_CREATE message. First the menu handle is obtained; we need it for checking menu items, as we saw in Chapter 8. Then we open a new kind of device context.

The Window Device Context

In WM_CREATE we open a micro PS with GpiCreatePS, and open a window device context with a new function, WinOpenWindowDC. Why not use a cached micro PS as we've done in previous programs? We need to specify a font during WM_CREATE, but this same font must be available during WM_PAINT processing, so we can use it to draw text on the window. The font is stored in the PS, so the PS must be kept in existence, not just during WM_PAINT processing as in previous examples, but for the duration of the program. A cached micro PS stays in existence only during processing of the WM_PAINT message, so we can't use it here. We need a micro PS. But before we obtain a micro PS, we need a device context to associate it with. (A normal PS would work, but is unnecessary.)

In the last chapter we used DevOpenDC to obtain a device context for the printer. Here we want to draw on the screen, or more specifically in our client window. The WinOpenWindowDC function opens a device context for a window.

Open Window Device Context

```
HDC WinOpenWindowDC(hwnd)
HWND hwnd     Window handle

Returns: Device context handle
```

We use the device context handle obtained from this function as an argument to GpiCreatePS, to obtain a micro PS.

Note that, unlike other devices, a window device context should not be closed, so DevCloseDC should not be called in this example.

The logical font is set, as it was in the last example, by filling in the FATTRS structure and executing GpiCreateLogFont and GpiSetCharSet. The font is set to Courier, which we assume is already in the system.

Setting the Character Mode

Next the character mode (described earlier) is set with the GpiSetCharMode function. The character mode is one of the character attributes.

Set Character Mode

```
BOOL GpiSetCharMode(hps, flMode)
HPS hps             Handle to presentation space
LONG flMode         Character mode

Returns: GPI_OK if successful, GPI_ERROR if error
```

The second argument to this function specifies the character mode. The possibilities are

Identifier	Character Mode
CM_DEFAULT	Default mode
CM_MODE1	Image font; only direction operates
CM_MODE2	Image font; size and orientation changes
CM_MODE3	Vector font; all character attributes work

WM_PAINT Processing

In previous examples the second argument to WinBeginPaint was set to NULL, and the function was used to return a handle to a cached micro PS. In FONTS, however, the presentation space is already allocated when we receive WM_PAINT, so WinBeginPaint doesn't need to obtain a PS. It is used only to set up the existing PS to update the invalid region. In this case, its second argument is set to the handle of the micro PS already obtained with GpiCreatePS.

Once the presentation space is made available with WinBeginPaint, the text "Testing 123" can be drawn on the window. This is done with the GpiCharStringAt function.

Draw Character String at Specified Position

```
LONG GpiCharStringAt(hps, pptlStart, cchString, pchString)
HPS hps             Handle to presentation space
PPOINTL pptlStart   Starting point
LONG cchString      Number of characters in pchString
PCH pchString       Character string

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

The second argument to this function is the starting point of the text string, which is expressed in world coordinates and stored in a structure of type POINTL. The string is stored in *pchString,* and the number of characters in *cchString.*

Another function, GpiCharString, is similar to GpiCharStringAt, except that it starts printing at the current position, rather than at a position specified by an argument. Two other calls, GpiCharStringPos and GpiCharStringPosAt, also display text, but allow more control of positioning and other text properties. We'll examine this approach in the next example.

Processing the WM_COMMAND Message

In the WM_COMMAND message we respond to the menu selections made by the user. There are two major sections, each one a *switch* construction starting with

```
switch (COMMANDMSG(&msg) -> cmd)
```

In the first of these sections, variables are set to values resulting from specific menu selections. For example, if the Helvetica font is selected from the "Fonts" menu, then "Helv" is copied into the *szFacename* member of the FATTRS structure. In the second section, the *case* statements for all the items in a menu are grouped together, and the appropriate action is taken. For instance, no matter what font is selected from the "Fonts" menu, a new font is set. The structure of the program could have been changed to use a single *switch* statement, but the resulting code would have been more complex.

Let's look at the operations that take place during WM_COMMAND processing.

Setting the Font

We already noted how the face name was set in the FATTRS structure as the result of font selection. Setting the font is handled similarly to the last example, using GpiCreateLogFont and GpiSetCharSet. Then (in the *if* statement) the menu item for the old font is unchecked and that for the new font is checked.

Setting the Font Characteristics

Font characteristics are specified by setting the *fsSelection* member of the FATTRS structure. The possibilities are

Identifier	**Font Characteristic**
FATTR_SEL_ITALIC	Italic
FATTR_SEL_BOLD	Bold
FATTR_SEL_STRIKEOUT	Strikeout (crossed out)
FATTR_SEL_UNDERSCORE	Underscore (underlined)

As it was to set the font, the selection is made in the first *switch* section and executed in the second section when the new logical font is created.

The GpiCreateLogFont function uses this argument to determine how to alter the font it has just loaded. It slants characters to make them italic, draws heavier strokes to make them bold, and so on. The resulting font is a combination of the font loaded from the DLL and the characteristics added by GpiCreateLogFont.

Checking menu items is slightly different for selecting characteristics than for selecting fonts. Since multiple characteristics can be checked simultaneously (text can be bold and italic, for example), the program (in the *else* statement) first finds out whether the menu item is checked or unchecked, and then sends a message telling it to assume the opposite state.

Setting the Character Box Size

Menu selections allow the character box size to be incremented or decremented in the X and Y directions. This involves changing the values of a SIZEF structure, *sizfxCharBox* in our example. This structure is then, in the second *switch* section, used as input to the GpiSetCharBox function. This function changes the size of the character box.

Change Size of Character Box

```
BOOL GpiSetCharBox(hps, psizfxBox)
HPS hps             Handle to presentation space
PSIZEF psizfxBox    New character box width and height

Returns: GPI_OK if successful, GPI_ERROR if error
```

The second argument to this function is a pointer to a structure of type SIZEF, defined this way in PMGPI.H:

```
typedef struct _SIZEF  {     /* sizfx */
   FIXED cx;                 /* width of rectangle (world coordinates) */
   FIXED cy;                 /* height of rectangle (world coordinates) */
   } SIZEF;
```

The new width and height are converted to the FIXED type and placed in the *sizfxCharBox* variable.

Setting the Character Box Angle

Menu selections allow the character box angle to be changed. The angle is specified by a point through which a line is drawn from the origin. Either the X or Y coordinate of the point can be incremented or decremented. These values are stored in the *gradl* variable, which is of type GRADIENTL. This structure is then used as input to the GpiSetCharAngle function, which changes the character box angle.

Change Angle of Character Box

```
BOOL GpiSetCharAngle(hps, pgradlAngle)
HPS hps                  Handle to presentation space
PGRADIENTL pgradlAngle   Endpoint (defines character angle)

Returns: GPI_OK if successful, GPI_ERROR if error
```

The second argument to this function is a pointer to the GRADIENTL structure, defined this wav in PMGPI.H:

```
typedef struct _GRADIENTL  {    /* gradl */
   LONG x;           /* x coordinate of endpoint */
   LONG y;           /* y coordinate of endpoint */
} GRADIENTL;
```

The point defined by this structure is the endpoint of a line that starts at the origin (0,0). By default the line is horizontal. Note that changing the X value of the endpoint has no effect while the Y value is 0, which it is when you start the program.

Setting the Character Box Shear

Menu selections permit the shear values to be changed. The shear is specified by an angle, which is defined by a point through which a line is drawn from the origin. Either the X or Y coordinate of the point can be incremented or decremented. These values are stored in the *ptlShear* variable, which is of type POINTL. This structure is used as an argument to the GpiSetCharShear function, which changes the character box shear.

Change Character Box Shear

```
BOOL GpiSetCharShear(hps, pptlShear)
HPS hps           Handle to presentation space
PPOINTL pptlShear Shear angle

Returns: GPI_OK if successful, GPI_ERROR if error
```

We've seen the POINTL structure, used in the second argument to this function, several times before. In this case this structure defines the endpoint of a line that starts at the origin. The angle that this line makes with the vertical (the Y axis) determines the shear angle. The default angle is 0. Note that changing the X value of the endpoint has no effect on the angle while the Y value is 0.

POSITIONING CHARACTERS WITHIN TEXT

Ordinarily PM positions characters automatically within a line of text. However, it's possible to specify a horizontal increment between each pair of characters, thus controlling character spacing.

We'll demonstrate this capability by showing how to justify text. *Justification* is arranging the characters in a line of text so the line has a certain length. Typically all the text lines in a document are made the same length to provide a neat look at both margins.

In PM, space can be added not only between words (which is the only justification typewriters and some printers can do), but between characters. This space can be added in pixel increments, so that there appears to be uniform spacing from character to character in the line of text. This is called *micro justification.*

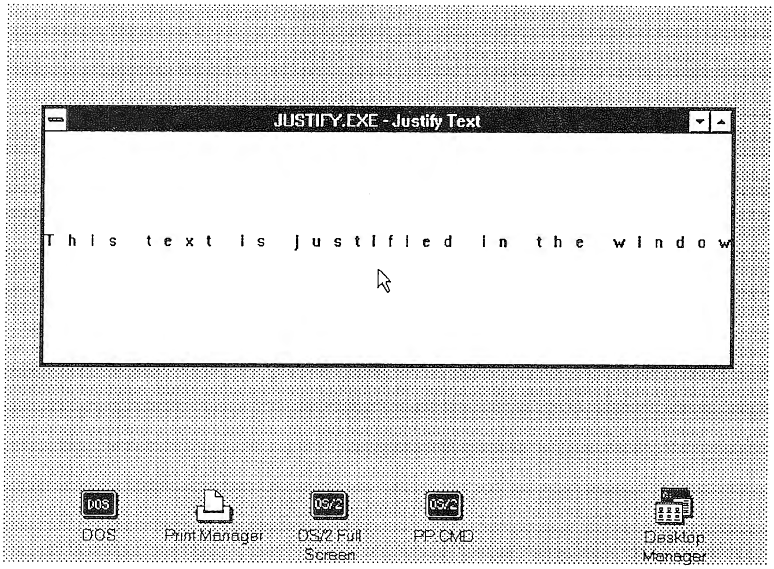


Figure 11-8. Output of the JUSTIFY program

In our example we'll show how to justify a single line of text. This procedure involves two new APIs: GpiQueryCharStringPos and GpiCharStringPos. The example prints a line of text so that it fits exactly between the left and right edges of the window, as shown in Figure 11-8. If you change the size of the window, the text will rejustify itself so it always fits exactly between the left and right margins.

If you make the window too narrow, the text will be cut off on the right margin. In a real application this is the point where a word-wrap procedure might be invoked to break the line of text one word earlier so it would fit on the line.

Here are the listings for JUSTIFY.C, JUSTIFY.H, JUSTIFY, and JUSTIFY.DEF.

```
/* --------------------------------*/
/* JUSTIFY - Justify text in window */
/* --------------------------------*/

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include <stdio.h>
#include <string.h>
```

```
#include "justify.h"

USHORT cdecl main(void)
   {
   HAB hab;                              /* handle for anchor block */
   HMQ hmq;                              /* handle for message queue */
   QMSG qmsg;                            /* message queue element */
   HWND hwnd, hwndClient;                /* handles for windows */

                                         /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);            /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);      /* create message queue */

                                         /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                         /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
            szClientClass, " - Justify Text", 0L, NULL, 0, &hwndClient);

                                         /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);               /* destroy frame window */
   WinDestroyMsgQueue(hmq);              /* destroy message queue */
   WinTerminate(hab);                    /* terminate PM usage */

   return 0;
   }

                                         /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                     MPARAM mp2)
   {
   HPS hps;
   static POINTL ptlStart = {0, 0};
   static SHORT cx;
   static CHAR str[STRLEN] = "This text is justified in the window";
   POINTL px[STRLEN+1];
   LONG dx[STRLEN];
   LONG dxadd;
   int i;

   switch (msg)
      {

      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
         GpiErase(hps);
         GpiMove(hps, &ptlStart);
                                         /* get char positions */
         GpiQueryCharStringPos(hps, 0L, STRLEN, &str[0], NULL, &px[0]);

         dxadd = (cx - px[STRLEN].x) / (STRLEN - 1);
         if (dxadd < 0) dxadd = 0;
```

```
         for (i = 0; i < STRLEN; i++)  /* add spaces */
            dx[i] = px[i+1].x - px[i].x + dxadd;

         dxadd = cx - px[STRLEN].x - dxadd * (STRLEN - 1);
                                       /* add leftover spaces */
         if (dxadd > 0) for (i = 0; i < dxadd; dx[i++]++);

                                       /* display justified string */
         GpiCharStringPos(hps, NULL, CHS_VECTOR, STRLEN, &str[0], &dx[0]);

         WinEndPaint(hps);
         break;

      case WM_SIZE:
         ptlStart.y = SHORT2FROMMP(mp2) / 2;
         cx = SHORT1FROMMP(mp2);
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                        /* NULL / FALSE */
   }
```

```
/* --------- */
/* JUSTIFY.H */
/* --------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define STRLEN 36L
```

```
# -----------------
# JUSTIFY Make file
# -----------------

justify.obj: justify.c justify.h
   cl -c -G2s -W3 -Zp justify.c

justify.exe: justify.obj justify.def
   link /NOD justify,,NUL,os2 slibce,justify
```

```
; -----------
; JUSTIFY.DEF
; -----------

NAME        JUSTIFY WINDOWAPI

DESCRIPTION 'Justify text in window'

PROTMODE
STACKSIZE   4096
```

During WM_SIZE processing, the vertical center of the window and the width of the window are assigned to *ptlStart.y* and *cx*, respectively.

In WM_PAINT a cached micro PS is obtained with WinBeginPaint. All the major work takes place during processing of this message. We first find out where the characters would be drawn if we weren't going to worry about justification. The resulting character positions are placed in an array. The distance from one character to the next character along the line of text (the X direction if the text is horizontal) is the *inter-character interval*. We must calculate how much to add to each inter-character interval to expand the line of text to fit the window; that is, to justify it. Once we've found these intervals, we use them to draw the justified line of text.

Discovering Character Positions

The GpiQueryCharStringPos function can be used to discover where every character in a text string would be located as if the string were displayed normally. The string isn't actually displayed by this API, but the positions of the characters are copied into an array. The values in this array form the starting point for the justification process.

Determine Position of Each Character in a String

```
BOOL GpiQueryCharStringPos(hps, flOptions, cchString, pchString,
                           adx, aptl)
HPS hps              Handle to presentation space
ULONG flOptions      CHS_VECTOR=use increments in adx, or 0
LONG cchString       Length of string
PCH pchString        String
PLONG adx            Array of increments
PPOINTL aptl         Array for character positions (# of chars +1)

Returns: GPI_OK if successful, GPI_ERROR if error
```

Some of the arguments to this function aren't used in this example. The programmer can, if desired, specify the increments between characters. This is done by setting the *flOptions* argument to CHS_VECTOR and filling in the *adx* array with the increments. We want to find out the normal positions of the characters, so we don't use this option. Thus *flOptions* is set to 0 and *adx* to NULL.

The *cchString* argument is the length of the string, and *pchString* is a pointer to the string whose character positions we want to determine. These positions are returned in *aptl*, an array of type POINTL. Note that this array must be one larger than the number of characters. The last element corresponds to the character position just beyond the last character in the string.

Justifying the Text

To display the character string, we will use GpiCharStringPos, which requires not the positions of the characters, but the increments in X position from each character to the next. As we calculate these increments, we'll store them in the array *dx*, which is an argument to this function.

We'll calculate the increments in two passes. In the first pass we'll get it almost right, but we'll have some leftover pixels. In the second pass we'll distribute the leftover pixels.

A First Approximation

First the distance from the right end of the text (that is, the unjustified nondisplayed text) to the right edge of the window is calculated. This is

```
cx - px(STRLEN).x
```

This distance must be redistributed amongst all the characters, so it is divided by the number of inter-character spaces, STRLEN−1, to yield the amount to add to each character. The result is stored in *dxadd*. Note that the same increment is added to each character regardless of whether the character is visible or a space. The situation is shown in Figure 11-9.

For instance, there might be 70 pixels between the right end of the text and the right edge of the window, and 36 characters in the string. In this case we want to insert exactly two pixels between each pair of characters (there's one less inter-character space than there are characters).

To calculate the increment for a particular character, we subtract the X position for the character from the X position for the next character. This is the original increment. Then we add *dxadd* to arrive at the justified increment. The result is stored in the appropriate element of array *dx*.

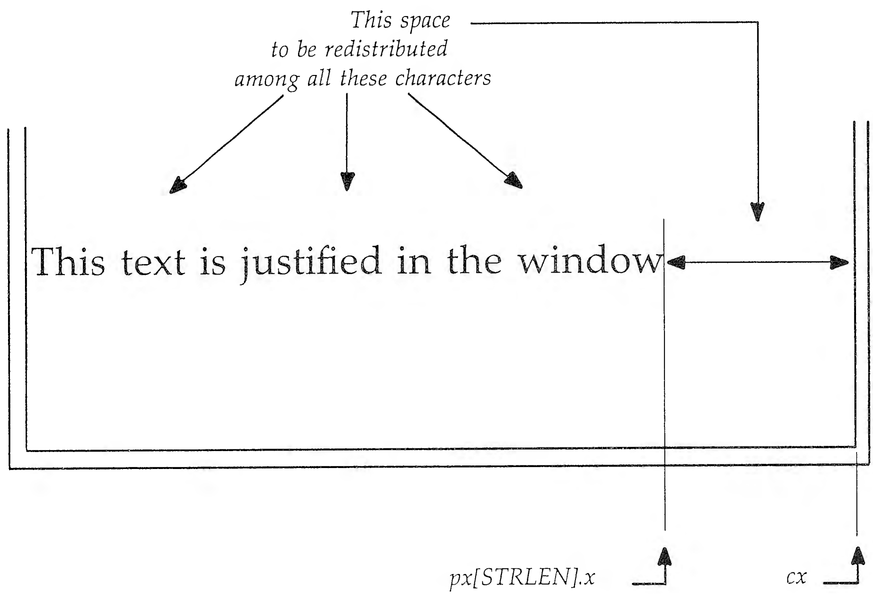


Figure 11-9. Justifying a line of text

If the window is narrower than the original line of text, *dxadd* will be negative. When this happens, we set *dxadd* to 0 so that no change is made to the increments when *dx* is calculated. In this case the text will have normal spacing and be clipped at the right end to fit the window. (We could have condensed the text by decreasing the normal intervals, but at some point the characters crowd each other and become unreadable.)

The Leftover Pixels

You might think at this point that we are done, and the string is ready to be displayed. Regrettably, our calculation of the increments is not quite exact. To determine *dxadd,* we divided the distance from the right end of the text to the right edge of the window by the number of inter-character spaces. We did this using integer arithmetic, since all the relevant variables are integers. Alas, this does not furnish an exact result, since in the majority of cases there will be a remainder from this division. For instance, if it is 40 pixels from the right edge of the text to the right edge of the window, and there are 36 characters in the string, we divided 40 by 35 and obtained a

value of 1 pixel per character. But there are 5 leftover pixels. If we don't account for them, the text will be 5 pixels too short. They must be inserted somewhere in the line of text.

Our approach to distributing these leftover pixels is to place them in the inter-character spaces at the beginning of the text string, until we run out of pixels. Thus if there are 5 leftover pixels, we'll increase the increment for the first 5 inter-character spaces by 1 (incrementing the increment).

How many leftover pixels are there? We already found the number of pixels that should be added to each interval if we ignore the leftover pixels. This is the value stored in *dxadd*. If we multiply this by the number of inter-character spaces, we get the number of pixels we've disposed of so far. This is

```
dxadd * (STRLEN - 1)
```

If we subtract this from the number of pixels between the right end of the text and the right edge of the window, we arrive at the number of leftover pixels. We use *dxadd* again to store this new value.

If this number is negative (because the line is longer than the window) or zero (because there are no leftover pixels), we skip the next step. Otherwise we go through a loop for each pixel and increment successive values of *dx* until all the leftover pixels are used up.

Now the intervals are correct to within a pixel, and the displayed text will exactly fill the width of the window.

Displaying the Text

The text is displayed using GpiCharStringPos.

Draw Character String from Current Position

```
LONG GpiCharStringPos(hps, prcl, flOptions, cchString, pchString, adx)
HPS hps             Handle to presentation space
PRECTL prcl         Coordinates of clipping rectangle
ULONG flOptions     CHS_CLIP, CHS_VECTOR, etc.
LONG cchString      Number of characters in string
PCH pchString       String to be displayed
PLONG adx           Array of increment values

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

This function can draw text with several different options. These are determined by the *flOptions* argument, which can be given the following values:

Identifier	Option
CHS_CLIP	Clip string to specified rectangle
CHS_LEAVEPOS	Reset current position to start of string
CHS_OPAQUE	Fill rectangle with background color
CHS_VECTOR	Use increments from *adx*

If CHS_CLIP is used, the *prcl* argument points to a rectangle to which the text will be clipped. We don't use this option, so *prcl* is NULL. If CHS_VECTOR is used, the characters will be located according to the intervals stored in the *adx* array. This is the option we want. The string *pchString,* which is *cchString* characters long, will be displayed, with the letters separated by the intervals in the *dx* array.

This approach can be extended to justify entire paragraphs and documents.

ADVANCED VIDEO INPUT/OUTPUT (AVIO)

Advanced video input/output, or AVIO, allows the creation of a special text-only presentation space. No graphics can be used, and the screen is divided into rows and columns of fixed-sized character positions, as it is in MS-DOS and OS/2 1.0 character-mode programs. However, an application that uses AVIO is a normal PM application, in that it processes messages like any other PM program.

Using AVIO is similar to using a CGA video buffer in text mode. The presentation space is composed of cells that contain the character's code and attributes. You can't control text position on a pixel basis as you can in other presentation spaces, and you can't change the size or appearance of the text or use different fonts.

Displaying text in an AVIO PS is not done with Gpi calls, but with Vio kernel calls, like VioWrtTTy and VioWrtCharStr.

One possible reason to use AVIO is to port existing text-only OS/2 1.0 programs to PM. The process is in theory simplified by using AVIO, since

the same Vio calls that worked in 1.0 will work in AVIO. However, the bulk of the program must still be rewritten, since the program architecture in PM is message based rather than procedure based. Given the amount of effort necessary for this translation, you might as well go the rest of the way and convert to a PM graphics-based text display, rather than use AVIO. Due to AVIO'S limited utility, we won't describe it further.

12

AREAS, PATHS, REGIONS, AND BITMAPS

This chapter discusses graphics objects that are somewhat more complex than the primitives described in Chapter 10. *Areas* are two-dimensional shapes, outlined by lines and arcs, that can be filled with colors and patterns. *Paths* are constructed from lines and arcs, and can be used for clipping and to create thick lines. *Regions* are rectangles, or combinations of rectangles, that are used for clipping and other purposes. *Bitmaps* are graphics images constructed from pixels, rather than from graphics primitives like lines and arcs. We'll cover these objects in turn, and in the last section we'll show how regions and paths can be used for clipping.

Along the way this chapter will explore other topics, including mix modes, patterns, fill modes, bounding rectangles, and bit-blitting.

AREAS

In Chapter 10 we used lines and arcs to produce shapes like rectangles, pie slices, and ellipses. How do we color these shapes, or fill them with a pattern? One answer is the *area*. An area can be defined as the interior of a closed series of lines and arcs. Once defined, the area can be filled with a solid color or a pattern.

Areas can be created in two ways. The most general is to use an *area bracket.* This permits the creation of arbitrary shapes. Also, special cases of areas can be created with the GpiBox and GpiFullArc functions. We'll examine these approaches in turn.

Areas with Area Brackets

A series of Gpi line and arc functions can be used to define an area. The series of functions is delimited by a GpiBeginArea and GpiEndArea pair. For example, the following code fragment defines an area consisting of a triangle whose vertices are *ptl1, ptl2,* and *ptl3:*

```
GpiBeginArea(hps, BA_ALTERNATE);  /* begin area bracket */
GpiMove(hps, &ptl1);              /* start at point 1 */
GpiLine(hps, &ptl2);              /* go to point 2 */
GpiLine(hps, &ptl3);              /* on to point 3 */
GpiLine(hps, &ptl1);              /* back to point 1 */
GpiEndArea(hps);                  /* end area bracket */
```

This sequence of Gpi functions is called an *area bracket.*

The last GpiLine call is not really necessary. When the GpiEndArea function is executed, PM automatically closes any figure that isn't already closed, drawing a line from the current position to the starting point. We include this call here for clarity.

Start an Area Bracket

```
BOOL GpiBeginArea(hps, flOptions)
HPS hps           Handle to presentation space
ULONG flOptions   Boundary and fill options

Returns: GPI_OK if successful, GPI_ERROR if error
```

The boundary and fill options in the *flOptions* argument specify whether the boundary of the area will be drawn, and the algorithm used to fill complex areas. Here are the identifiers used for these options:

Identifier	**Boundary and Fill Options**
BA_BOUNDARY	Boundary lines enclose area

BA_NOBOUNDARY	No boundary lines enclose area (default)
BA_ALTERNATE	Fills interior in alternate mode (default)
BA_WINDING	Fills interior in winding mode

The boundary line is one pixel wide, if drawn. We'll explore the fill modes at the end of this section.

End an Area Bracket

```
LONG GpiEndArea(hps)
HPS hps           Handle to presentation space

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

Our example uses area brackets to fill the slices in the pie chart created in the PIE example from the last chapter. The large slice is colored dark green, and the small slice is colored red. Figure 12-1 shows the result (unfortunately in black and white).

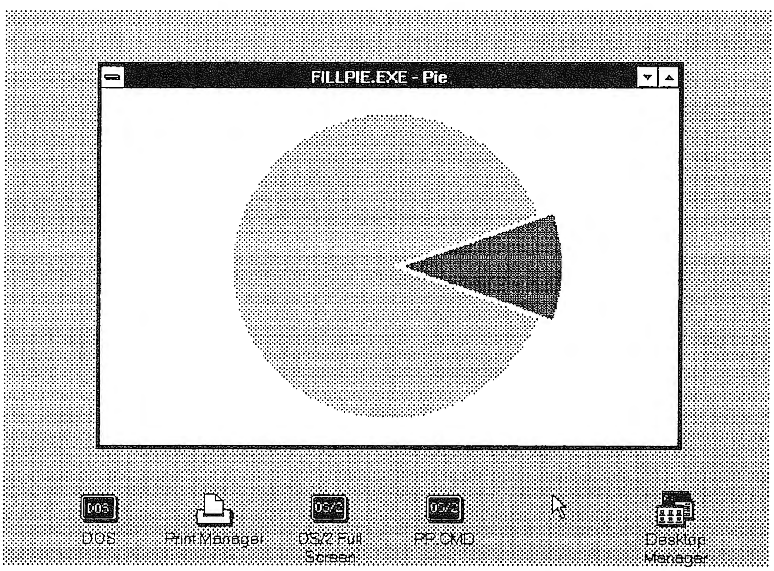


Figure 12-1. Output of the FILLPIE program

Here are the listings for FILLPIE.C, FILLPIE.H, FILLPIE, and FILLPIE. DEF.

```
/* ---------------------------------- */
/* FILLPIE.C - Draw a filled pie chart */
/* ---------------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include <stdlib.h>

#include "fillpie.h"

USHORT cdecl main(void)
   {
   HAB hab;                              /* handle for anchor block */
   HMQ hmq;                              /* handle for message queue */
   QMSG qmsg;                            /* message queue element */
   HWND hwnd, hwndClient;                /* handles for windows */

                                         /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);            /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);      /* create message queue */

                                         /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                         /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
            szClientClass, " - Pie", 0L, NULL, 0, &hwndClient);

                                         /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);               /* destroy frame window */
   WinDestroyMsgQueue(hmq);              /* destroy message queue */
   WinTerminate(hab);                    /* terminate PM usage */

   return 0;
   }

                                         /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   HPS hps;
   static POINTL ptlCenter;
   static FIXED fxMult;
   static SIZEL sizl = {0, 0};
   switch (msg)
```

```
         {
         case WM_PAINT:
            hps = WinBeginPaint(hwnd, NULL, NULL);
            GpiSetPS(hps, &sizl, PU_LOENGLISH);
                                              /* draw pie */
            GpiMove (hps, &ptlCenter);
            GpiSetColor(hps, CLR_DARKGREEN);
            GpiBeginArea(hps, BA_BOUNDARY);
            GpiPartialArc(hps, &ptlCenter, fxMult, MAKEFIXED(20,0),
               MAKEFIXED(320,0));
            GpiLine(hps, &ptlCenter);
            GpiEndArea(hps);
                                              /* draw slice */
            ptlCenter.x += FIXEDINT(fxMult) / 10;
            GpiMove(hps, &ptlCenter);
            GpiSetColor(hps, CLR_RED);
            GpiBeginArea(hps, BA_BOUNDARY);
            GpiPartialArc(hps, &ptlCenter, fxMult, MAKEFIXED(340,0),
               MAKEFIXED(40,0));
            GpiEndArea(hps);
            WinEndPaint(hps);
            break;

         case WM_SIZE:                        /* set new pie size */
            ptlCenter.x = SHORT1FROMMP(mp2) / 2;
            ptlCenter.y = SHORT2FROMMP(mp2) / 2;
            hps = WinGetPS(hwnd);
            GpiSetPS(hps, &sizl, PU_LOENGLISH);
            GpiConvert(hps, CVTC_DEVICE, CVTC_PAGE, 1L, &ptlCenter);
            fxMult = MAKEFIXED(min(min(ptlCenter.x, ptlCenter.y) * .85, 255),
                               0);
            WinReleasePS(hps);
            break;

         case WM_ERASEBACKGROUND:
            return TRUE;
            break;

         default:
            return(WinDefWindowProc(hwnd, msg, mp1, mp2));
            break;
         }

      return NULL;                            /* NULL / FALSE */
      }
```

```
/* --------- */
/* FILLPIE.H */
/* --------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);
```

```
# -----------------
# FILLPIE Make file
# -----------------
```

```
fillpie.obj: fillpie.c fillpie.h
   cl -c -G2s -W3 -Zp fillpie.c

fillpie.exe: fillpie.obj fillpie.def
   link /NOD fillpie,,NUL,os2 slibce,fillpie
```

```
; -----------
; FILLPIE.DEF
; ------------

NAME        FILLPIE      WINDOWAPI

DESCRIPTION 'Draw filled pie chart'

PROTMODE
STACKSIZE   4096
```

Two area brackets are created in this program. In the first, the partial arc is drawn for the large slice with GpiPartialArc. Remember that this function starts by drawing a line from the center to the beginning of the arc, and then draws the arc itself. To complete a closed area, we draw the line back from the end of the arc to the center. GpiSetColor sets the color before the area bracket is defined, so the area is filled with this color as soon as it is created.

The second area bracket is similar, except that we left out the GpiLine function that completes the area. The slice is drawn and filled anyway, demonstrating that the GpiEndArea will indeed close an area if necessary.

Areas with GpiBox and GpiFullArc

In the last chapter we used GpiBox and GpiFullArc with the fill argument set to DRO_OUTLINE. We can create filled areas simply by setting this argument to DRO_FILL, or, if we want the outline as well as the area, DRO_OUTLINEFILL.

In the next example, MIXMODES, we demonstrate this capability, and at the same time continue our exploration of color mix modes, which we touched on in the MARKER example in Chapter 10.

Our example program draws a circle in the client window wherever the user clicks mouse button 1. The circle is drawn using GpiFullArc, with the *flFlags* argument set to DRO_OUTLINEFILL.

There are two submenus in MIXMODES, titled "Colors" and "Mix". The selection from "Colors" determines the color used to fill the circle, and the selection from "Mix" determines how this color will be combined with the

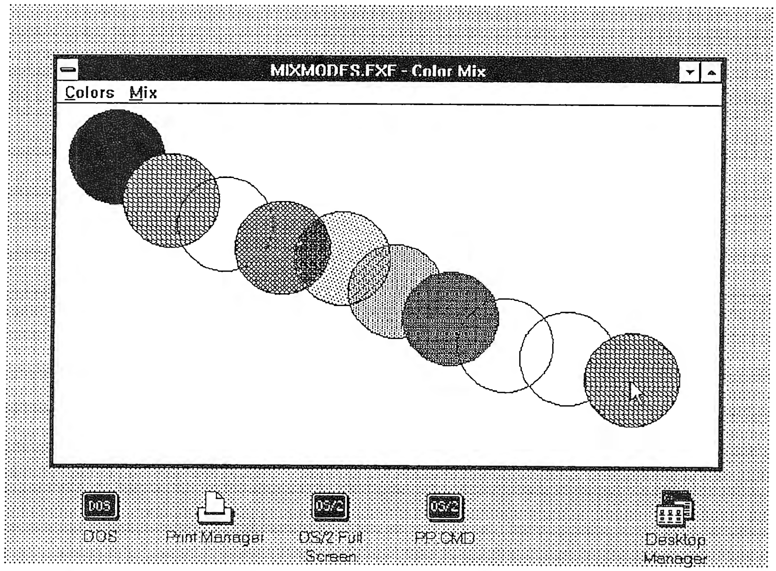


Figure 12-2. Output of the MIXMODES program

colors under the circle. By selecting different colors and mix modes, and overlapping one circle on another, you can explore the hundreds of possible combinations. A sample is shown in Figure 12-2.

Here are the listings for MIXMODES.C, MIXMODES.H, MIXMODES, MIXMODES.DEF, and MIXMODES.RC:

```
/* --------------------------- */
/* MIXMODES.C - Mix modes demo */
/* --------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "mixmodes.h"

USHORT cdecl main(void)
   {
   HAB hab;                           /* handle for anchor block */
   HMQ hmq;                           /* handle for message queue */
   QMSG qmsg;                         /* message queue element */
   HWND hwnd, hwndClient;             /* handles for windows */

                                      /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_MENU |
                         FCF_TASKLIST;
```

```
   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);            /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);      /* create message queue */

                                         /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                         /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Color Mix", 0L, NULL, ID_FRAMERC,
             &hwndClient);

                                         /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);               /* destroy frame window */
   WinDestroyMsgQueue(hmq);              /* destroy message queue */
   WinTerminate(hab);                    /* terminate PM usage */

   return 0;
   }

                                         /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   HPS hps;
   static ARCPARAMS arcp = {1, 1, 0, 0};
   static FIXED fxMult = MAKEFIXED(150, 0);
   static SIZEL sizl = {0 , 0};
   static POINTL ptl[MAXCIRCLES];
   static USHORT usMix[MAXCIRCLES];
   static LONG clr[MAXCIRCLES];
   static int i = 0;
   int j;
   static AREABUNDLE abnd = {CLR_NEUTRAL, 0L, FM_OVERPAINT};
   static HWND hwndMenu;

   switch (msg)
      {
      case WM_COMMAND:
         switch (COMMANDMSG(&msg) -> cmd)
            {
                                         /* change color */
            case ID_BACKGROUND:
            case ID_BLUE:
            case ID_RED:
            case ID_PINK:
            case ID_GREEN:
            case ID_CYAN:
            case ID_YELLOW:
            case ID_NEUTRAL:

                                         /* un-check old color */
               WinSendMsg(hwndMenu, MM_SETITEMATTR,
                  MPFROM2SHORT((SHORT)abnd.lColor + ID_COLORS, TRUE),
                  MPFROM2SHORT(MIA_CHECKED, ~MIA_CHECKED));
```

```
                                    /* check new color */
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
               MPFROM2SHORT(COMMANDMSG(&msg) -> cmd, TRUE),
               MPFROM2SHORT(MIA_CHECKED, MIA_CHECKED));

            abnd.lColor = (COMMANDMSG(&msg) -> cmd) - ID_COLORS;
            break;

                                    /* change mix mode */
         case ID_OR:
         case ID_OVERPRINT:
         case ID_LEAVEALONE:
         case ID_XOR:
         case ID_AND:
         case ID_SUBTRACT:
         case ID_MASKSRCNOT:
         case ID_ZERO:
         case ID_NOTMERGESRC:
         case ID_NOTXORSRC:
         case ID_INVERT:
         case ID_MERGESRCNOT:
         case ID_NOTCOPYSRC:
         case ID_MERGENOTSRC:
         case ID_NOTMASKSRC:

                                    /* un-check old mode */
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
               MPFROM2SHORT(abnd.usMixMode + ID_MIXMODES, TRUE),
               MPFROM2SHORT(MIA_CHECKED, ~MIA_CHECKED));

                                    /* check new mode */
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
               MPFROM2SHORT(COMMANDMSG(&msg) -> cmd, TRUE),
               MPFROM2SHORT(MIA_CHECKED, MIA_CHECKED));

            abnd.usMixMode = (COMMANDMSG(&msg) -> cmd) - ID_MIXMODES;
            break;
         }
      break;

   case WM_BUTTON1DOWN:
      if (i <= MAXCIRCLES)
         {                          /* add new circle */
         ptl[i].x = SHORT1FROMMP(mp1);
         ptl[i].y = SHORT2FROMMP(mp1);

         hps = WinGetPS(hwnd);
         GpiSetPS(hps, &sizl, PU_LOMETRIC);
         GpiSetArcParams(hps, &arcp);
         GpiConvert(hps, CVTC_DEVICE, CVTC_PAGE, 1L, &ptl[i]);
         clr[i] = abnd.lColor;
         usMix[i] = abnd.usMixMode;
                                    /* set color & mix mode */
         GpiSetAttrs(hps, PRIM_AREA, ABB_COLOR | ABB_MIX_MODE,
            0L, &abnd);
         GpiMove(hps, &ptl[i++]);
         GpiFullArc(hps, DRO_OUTLINEFILL, fxMult);
         WinReleasePS(hps);
         }
      else
```

```
            WinAlarm(HWND_DESKTOP, WA_WARNING);

         WinDefWindowProc(hwnd, msg, mp1, mp2);
         return TRUE;
         break;

      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
         GpiSetPS(hps, &sizl, PU_LOMETRIC);
         GpiSetArcParams(hps, &arcp);
                                         /* redraw all circles */
         for (j = 0; j < i; j++)
            {
            abnd.lColor = clr[j];
            abnd.usMixMode = usMix[j];
            GpiSetAttrs(hps, PRIM_AREA, ABB_COLOR | ABB_MIX_MODE,
               0L, &abnd);
            GpiMove (hps, &ptl[j]);
            GpiFullArc(hps, DRO_OUTLINEFILL, fxMult);
            }
         WinEndPaint(hps);
         break;

      case WM_CREATE:
                                         /* get menu window handle */
         hwndMenu = WinWindowFromID(WinQueryWindow(hwnd, QW_PARENT,
                                         FALSE),FID_MENU);
         break;

      case WM_ERASEBACKGROUND:
         return TRUE;
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                          /* NULL / FALSE */
   }
```

```
/* ---------- */
/* MIXMODES.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1

#define ID_COLORS 200
#define ID_BACKGROUND 200
#define ID_BLUE 201
#define ID_RED 202
#define ID_PINK 203
#define ID_GREEN 204
#define ID_CYAN 205
```

```
#define ID_YELLOW 206
#define ID_NEUTRAL 207

#define ID_MIXMODES 300

#define ID_OR 301
#define ID_OVERPRINT 302
#define ID_LEAVEALONE 305
#define ID_XOR 304
#define ID_AND 306
#define ID_SUBTRACT 307
#define ID_MASKSRCNOT 308
#define ID_ZERO 309
#define ID_NOTMERGESRC 310
#define ID_NOTXORSRC 311
#define ID_INVERT 312
#define ID_MERGESRCNOT 313
#define ID_NOTCOPYSRC 314
#define ID_MERGENOTSRC 315
#define ID_NOTMASKSRC 316
#define ID_ONE 317

#define MAXCIRCLES 30
```

```
# ------------------
# MIXMODES Make file
# ------------------

mixmodes.obj: mixmodes.c mixmodes.h
   cl -c -G2s -W3 -Zp mixmodes.c

mixmodes.res: mixmodes.rc mixmodes.h
   rc -r mixmodes.rc

mixmodes.exe: mixmodes.obj mixmodes.def
   link /NOD mixmodes,,NUL,os2 slibce,mixmodes
   rc mixmodes.res

mixmodes.exe: mixmodes.res
   rc mixmodes.res
```

```
; ------------
; MIXMODES.DEF
; ------------

NAME         MIXMODES      WINDOWAPI

DESCRIPTION 'Mix modes'

PROTMODE
STACKSIZE   4096
```

```
/*------------ */
/* MIXMODES.RC */
/*------------ */
```

```
#define INCL_GPI
#include <os2.h>

#include "mixmodes.h"

MENU ID_FRAMERC
   BEGIN
      SUBMENU "~Colors", ID_COLORS
         BEGIN
            MENUITEM "BACKGROUND", ID_BACKGROUND
            MENUITEM "BLUE", ID_BLUE
            MENUITEM "RED", ID_RED
            MENUITEM "PINK", ID_PINK
            MENUITEM "GREEN", ID_GREEN
            MENUITEM "CYAN", ID_CYAN
            MENUITEM "YELLOW", ID_YELLOW
            MENUITEM "NEUTRAL", ID_NEUTRAL, , MIA_CHECKED
         END

      SUBMENU "~Mix", ID_MIXMODES
         BEGIN
            MENUITEM "OR", ID_OR
            MENUITEM "OVERPRINT", ID_OVERPRINT, , MIA_CHECKED
            MENUITEM "LEAVEALONE", ID_LEAVEALONE
            MENUITEM "XOR", ID_XOR
            MENUITEM "AND", ID_AND
            MENUITEM "SUBTRACT", ID_SUBTRACT
            MENUITEM "MASKSRCNOT", ID_MASKSRCNOT
            MENUITEM "ZERO", ID_ZERO
            MENUITEM "NOTMERGESRC", ID_NOTMERGESRC
            MENUITEM "NOTXORSRC", ID_NOTXORSRC
            MENUITEM "INVERT", ID_INVERT
            MENUITEM "MERGESRCNOT", ID_MERGESRCNOT
            MENUITEM "NOTCOPYSRC", ID_NOTCOPYSRC
            MENUITEM "MERGENOTSRC", ID_MERGENOTSRC
            MENUITEM "NOTMASKSRC", ID_NOTMASKSRC
         END
   END
```

Setting the Color

When the user selects a color from the "Colors" menu, a CLR_ identifier is set in the *abnd* structure for use in the GpiSetAttrs function (described in the last chapter). The menu-selectable identifiers are

Identifier	**Color**
CLR_BACKGROUND	White (screen background)
CLR_BLUE	Blue
CLR_RED	Red
CLR_PINK	Magenta
CLR_GREEN	Green

CLR_CYAN	Cyan
CLR_YELLOW	Yellow
CLR_NEUTRAL	Black (screen neutral)

Setting the Mix Mode

Mix modes determine how the color of a graphics object interacts with the color already on the drawing surface. (The "drawing surface" can be the screen, a printer or a plotter.) For instance, what happens when you place a blue circle over a red circle? An obvious guess is that the overlap turns blue, the last color drawn; but there are many other possibilities. The resulting color might be red, the underlying color. Or it might be a mixture of the two colors, or another color altogether.

Colors are specified in the logical color table. To mix two colors, bitwise operations are performed on the index number for the object's color, and on the index of the drawing surface. The result is the final color index. Figure 12-3 shows how this looks.

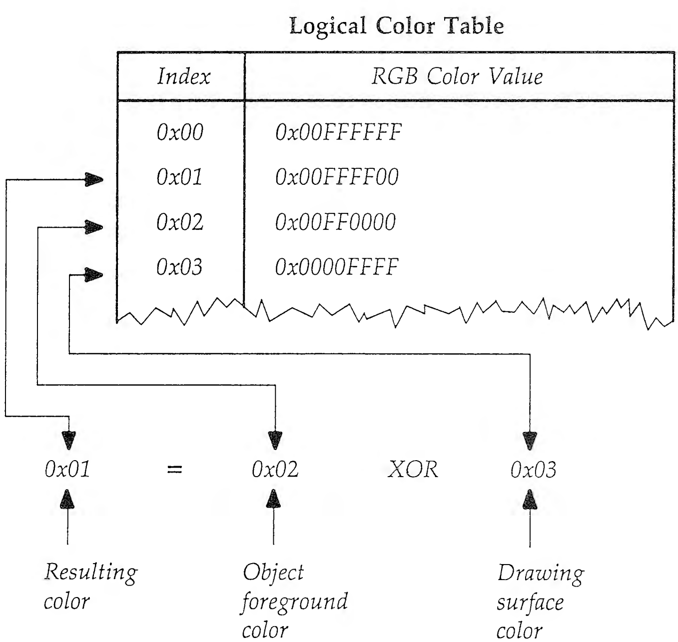


Figure 12-3. Mix modes and logical color tables

Areas, like lines and arcs, have only a foreground color, so when they overlap, only one mix mode applies. But, as we saw in Chapter 10, markers have both a foreground and a background color, each of which uses a separate mix mode. The character primitive also has both foreground and background colors.

There are 16 mix modes. Here are the mix mode identifiers, defined in PMGPI.H, for the foreground color:

Identifier	**Final Foreground Color**
FM_DEFAULT	Same as FM_OVERPAINT
FM_OR	OR foreground and drawing surface
FM_OVERPAINT	Always foreground color
FM_XOR	XOR foreground and drawing surface
FM_LEAVEALONE	Always drawing surface color
FM_AND	AND foreground and drawing surface
FM_SUBTRACT	Invert foreground, AND with drawing surface
FM_MASKSRCNOT	Invert drawing surface, AND foreground
FM_ZERO	Always 0
FM_NOTMERGESRC	Inverse of result from FM_OR
FM_NOTXORSRC	Inverse of result from FM_XOR
FM_INVERT	Inverse of drawing surface
FM_MERGESRCNOT	Invert drawing surface, AND foreground
FM_NOTCOPYSRC	Inverse of foreground
FM_MERGENOTSRC	Invert foreground, AND drawing surface
FM_NOTMASKSRC	Inverse of result from FM_AND
FM_ONE	Always 1

Note that a graphics object about to be drawn is often referred to as the *source,* and the underlying color of the drawing surface as the *destination.* Thus FM_NOTCOPYSRC means copy the foreground color of the object (the source) to the drawing surface (the destination), then take the inverse.

The 16 FM_ mix modes represent all possible ways to combine a two-color (for instance, black-and-white) object with a two-color drawing surface. Why? Assume a black pixel is 0 and a white pixel is 1. There are four combinations of the object and drawing surface colors: 00, 01, 10, and 11. A single mix mode specifies what each of these four combinations will yield; for instance, 00=0, 01=1, 10=0, 11=1. This result can be represented by the four binary digits 0101. There are 16 ways to arrange four binary digits: 0000, 0001, and so on up to 1111, so there can be a maximum of 16 mix modes for combining black and white.

Since mix modes use the *index values* of the logical color table, the color resulting from a particular mix mode depends on the RGB colors inserted

in the color table at these indices. By inserting different RGB values in the logical color table and using different mix modes, almost any conceivable effect can be achieved.

Although they aren't used in this example, here are the identifiers for the background mix modes:

Identifier	Final Background Color
BM_DEFAULT	Same as BM_LEAVEALONE
BM_OR	OR background and drawing surface
BM_OVERPAINT	Always background color
BM_XOR	XOR background and drawing surface
BM_LEAVEALONE	Always drawing surface color

Actually the same 16 FM_ mix modes can be used for the background as for foreground color, but only those shown here are given BM_ identifiers.

Using the MIXMODES program, you can experiment with the mix modes. The most common are probably overpaint, XOR, and OR. In overpaint the object color simply replaces the colors on the drawing surface.

XOR is useful for animation. If you XOR an object onto the drawing surface, and then XOR it again, it vanishes. You can try this by clicking repeatedly in the same place. You'll see that every two clicks brings back the underlying colors. This makes it easy to draw and redraw an object in different locations.

OR is useful when you want to emphasize the area where two graphics objects overlap; this area will be colored differently from either the object or the drawing surface.

Patterns

An area, whether generated with an area bracket or with GpiBox or GpiFullArc, can be filled with a *pattern*. Actually, the solid colors we used to fill the pie slices and the circles in the examples above are patterns: *solid* patterns. However, there are other possibilities. PM defines 16 standard patterns.

Our next example demonstrates these patterns by drawing a bar chart in which each of 16 bars is drawn in a different pattern. This is shown in Figure 12-4.

Here are the listings for PATTERNS.C, PATTERNS.H, PATTERNS, and PATTERNS.DEF.

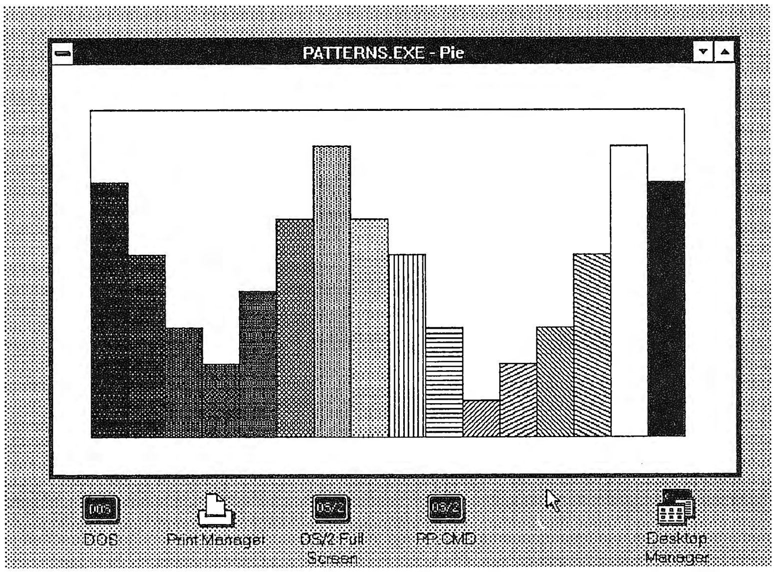


Figure 12-4. Output of the PATTERNS program

```
/* ---------------------------- */
/* PATTERNS.C - Draw a bar chart */
/* ---------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include <stdlib.h>

#include "patterns.h"

USHORT cdecl main(void)
   {
   HAB hab;                            /* handle for anchor block */
   HMQ hmq;                            /* handle for message queue */
   QMSG qmsg;                          /* message queue element */
   HWND hwnd, hwndClient;              /* handles for windows */

                                       /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);          /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);    /* create message queue */

                                       /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                       /* create standard window */
```

```
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
            szClientClass, " - Pie", 0L, NULL, 0, &hwndClient);

                                    /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);          /* destroy frame window */
   WinDestroyMsgQueue(hmq);         /* destroy message queue */
   WinTerminate(hab);               /* terminate PM usage */

   return 0;
   }

                                    /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                     MPARAM mp2)
   {
   HPS hps;
   static POINTL ptlUnit;
   POINTL ptl;
   static LONG lgraph[] = {7, 5, 3, 2, 4, 6, 8, 6, 5, 3, 1, 2, 3, 5, 8, 7};
   static LONG lSymbol[] = {PATSYM_DENSE1, PATSYM_DENSE2, PATSYM_DENSE3,
                            PATSYM_DENSE4, PATSYM_DENSE5, PATSYM_DENSE6,
                            PATSYM_DENSE7, PATSYM_DENSE8, PATSYM_VERT,
                            PATSYM_HORIZ, PATSYM_DIAG1, PATSYM_DIAG2,
                            PATSYM_DIAG3, PATSYM_DIAG4, PATSYM_NOSHADE,
                            PATSYM_SOLID};
   int i;

   switch (msg)
      {
      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
                                    /* draw border */
         ptl.x = ptlUnit.x; ptl.y = ptlUnit.y;
         GpiMove(hps, &ptl);
         ptl.x = ptlUnit.x * (NPOINTS + 1);
         ptl.y = ptlUnit.y * 10;
         GpiBox(hps,DRO_OUTLINE, &ptl, 0L, 0L);
         ptl.x = ptlUnit.x;

                                    /* draw bars */
         for (i = 0; i < NPOINTS; i++)
            {
            ptl.x += ptlUnit.x;
            ptl.y = lgraph[i] * ptlUnit.y + ptlUnit.y;
            GpiSetPattern(hps, lSymbol[i]);
            GpiBox(hps, DRO_OUTLINEFILL, &ptl, 0L, 0L);
            ptl.y = ptlUnit.y;
            GpiMove(hps, &ptl);
            }

         WinEndPaint(hps);
         break;

      case WM_SIZE:
         ptlUnit.x = SHORT1FROMMP(mp2) / (NPOINTS + 2);
         ptlUnit.y = SHORT2FROMMP(mp2) / 11;
         break;
```

```
      case WM_ERASEBACKGROUND:
         return TRUE;
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                        /* NULL / FALSE */
   }
```

```
/* ---------- */
/* PATTERNS.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define NPOINTS 16
```

```
# ------------------
# PATTERNS Make file
# ------------------

patterns.obj: patterns.c patterns.h
   cl -c -G2s -W3 -Zp patterns.c

patterns.exe: patterns.obj patterns.def
   link /NOD patterns,,NUL,os2 slibce,patterns
```

```
; ------------
; PATTERNS.DEF
; ------------

NAME        PATTERNS WINDOWAPI

DESCRIPTION 'Draw a bar chart'

PROTMODE
STACKSIZE   4096
```

The pattern is set with the GpiSetPattern function. Once set, all areas will be filled using this pattern until it's changed to something else.

Set Fill Pattern

```
BOOL GpiSetPattern(hps, lSymbol)
HPS hps              Handle to presentation space
LONG lSymbol         Pattern

Returns: GPI_OK if successful, GPI_ERROR if error
```

If the default pattern set is used, the *lSymbol* argument can be one of the following:

Identifier	Pattern
PATSYM_BLANK	Background (no fill)
PATSYM_DEFAULT	Device dependent (screen = white)
PATSYM_DENSE1	Mostly foreground
PATSYM_DENSE2	Less foreground
PATSYM_DENSE3	Less foreground
PATSYM_DENSE4	Mid-range
PATSYM_DENSE5	Mid-range
PATSYM_DENSE6	More background
PATSYM_DENSE7	More background
PATSYM_DENSE8	Mostly background
PATSYM_DIAG1	SW-NE lines, narrow spacing
PATSYM_DIAG2	SW-NE lines, wide spacing
PATSYM_DIAG3	NW-SE lines, narrow spacing
PATSYM_DIAG4	NW-SE lines, wide spacing
PATSYM_HALFTONE	Half foreground, half background
PATSYM_HORIZ	Horizontal lines
PATSYM_NOSHADE	Same as blank
PATSYM_SOLID	Same as default
PATSYM_VERT	Vertical lines

If you don't like these patterns, you can create your own custom patterns from bitmaps, using the GpiSetPatternSet function. We'll examine this possibility when we talk about bitmaps later in the chapter.

Fill Modes

If there's one area inside another, like the hole in a doughnut, and you fill the outer one, should the inner one be filled, too? That depends on what

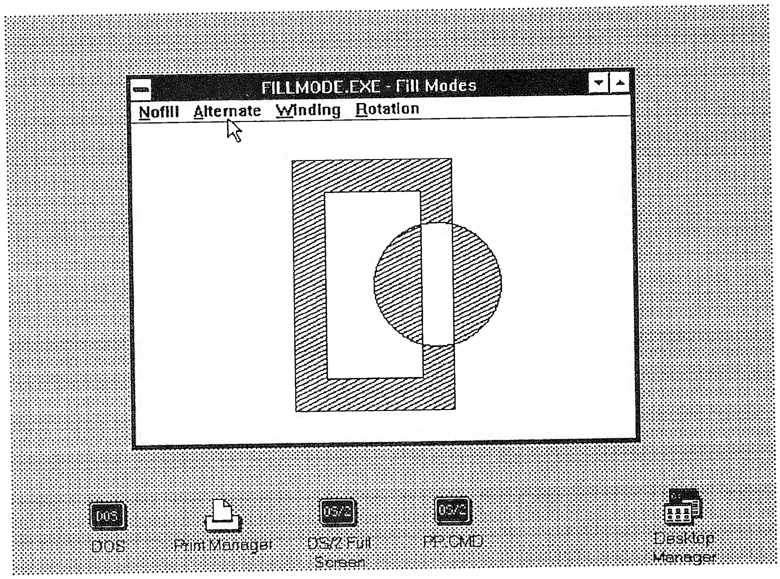


Figure 12-5. Output of the FILLMODE program

you're drawing. The hole in a doughnut should not be filled with the same color as the doughnut, but an aluminum rivet on an aluminum airplane wing probably should. More complex situations can arise: areas inside of areas inside of other areas, and areas formed by lines that cross each other.

Some programming environments perform the fill operation by flowing color into an area from a seed point, stopping at boundary lines. The questions just described don't arise with this simple approach. PM provides filling algorithms of greater power and flexibility.

There are two filling modes: *alternate* and *winding*. Our example program demonstrates these different modes. It draws two concentric boxes, with a circle overlapping both of them. Figure 12-5 shows the program when alternate fill mode has been selected.

Here are the listings for FILLMODE.C, FILLMODE.H, FILLMODE, FILLMODE.DEF, and FILLMODE.RC:

```
/* ------------------------------------------- */
/* FILLMODE.C - Using the different fill modes */
/* ------------------------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "fillmode.h"
```

```
USHORT cdecl main(void)
   {
   HAB hab;                              /* handle for anchor block */
   HMQ hmq;                              /* handle for message queue */
   QMSG qmsg;                            /* message queue element */
   HWND hwnd, hwndClient;                /* handles for windows */

                                         /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_MENU |
                         FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);            /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);      /* create message queue */

                                         /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                         /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Fill Modes", 0L, NULL, ID_FRAMERC,
             &hwndClient);

                                         /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);               /* destroy frame window */
   WinDestroyMsgQueue(hmq);              /* destroy message queue */
   WinTerminate(hab);                    /* terminate PM usage */

   return 0;
   }

                                         /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   HPS hps;
   HWND static hwndMenu;                 /* menu window handle */
   static SIZEL sizl = {0, 0};
   POINTL ptl;
   static LONG flBracketMode = -1L;
   static SHORT fDirection = ID_COUNTERCW;
   static POINTL aptlBox[] = {{600, 200}, {900, 200},
                              {900, 800}, {600, 800}};

   switch (msg)
      {
      case WM_COMMAND:
         {
         switch (COMMANDMSG(&msg) -> cmd)
            {
            case ID_NOFILL: flBracketMode = -1L; break;
            case ID_ALTERNATE: flBracketMode = BA_ALTERNATE; break;
            case ID_WINDING: flBracketMode = BA_WINDING; break;

            case ID_COUNTERCW:
            case ID_CLOCKWISE:
                                         /* un-check old menu item */
```

```
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
               MPFROM2SHORT(fDirection, TRUE),
               MPFROM2SHORT(MIA_CHECKED, ~MIA_CHECKED));

            fDirection = COMMANDMSG(&msg) -> cmd;

                                    /* check new menu item */
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
               MPFROM2SHORT(fDirection, TRUE),
               MPFROM2SHORT(MIA_CHECKED, MIA_CHECKED));
            break;
         }
      WinInvalidateRect(hwnd, NULL, FALSE);
      break;
      }

   case WM_PAINT:
      hps = WinBeginPaint(hwnd, NULL, NULL);
      GpiSetPS(hps, &sizl, PU_LOMETRIC);   /* low metric */
      GpiErase(hps);
      GpiSetPattern(hps, PATSYM_DIAG2);     /* set pattern */
      if (flBracketMode != -1L)      /* begin area bracket */
         GpiBeginArea(hps, BA_BOUNDARY | flBracketMode);

      ptl.x = 500; ptl.y = 100;      /* outer box */
      GpiMove(hps, &ptl);            /* counter-clockwise */
      ptl.x = 1000; ptl.y = 900;
      GpiBox(hps, DRO_OUTLINE, &ptl, 0L, 0L);

      if (fDirection == ID_COUNTERCW)
         {                           /* counter-clockwise */
         GpiMove(hps, &aptlBox[0]);
         GpiBox(hps, DRO_OUTLINE, &aptlBox[2], 0L, 0L);
         }
      else
         {                           /* clockwise */
         GpiMove(hps, &aptlBox[1]);
         GpiBox(hps, DRO_OUTLINE, &aptlBox[3], 0L, 0L);
         }

      ptl.x = 950; ptl.y = 500;      /* inner circle */
      GpiMove(hps, &ptl);            /* counter-clockwise */
      GpiFullArc(hps, DRO_OUTLINE, MAKEFIXED(200,0));

      if (flBracketMode != -1L)      /* end area bracket */
         GpiEndArea(hps);
      WinEndPaint(hps);
      break;

   case WM_CREATE:
                                     /* get menu window handle */
      hwndMenu = WinWindowFromID(WinQueryWindow(hwnd, QW_PARENT,
                                     FALSE),FID_MENU);

      break;

   default:
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }
```

```
    return NULL;                         /* NULL / FALSE */
    }
```

```
/* ---------- */
/* FILLMODE.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_NOFILL 101
#define ID_ALTERNATE 102
#define ID_WINDING 103
#define ID_DIRECTIONU 110
#define ID_COUNTERCW 111
#define ID_CLOCKWISE 112
```

```
# ------------------
# FILLMODE Make file
# ------------------

fillmode.obj: fillmode.c fillmode.h
   cl -c -G2s -W3 -Zp fillmode.c

fillmode.res: fillmode.rc fillmode.h
   rc -r fillmode.rc

fillmode.exe: fillmode.obj fillmode.def
   link /NOD fillmode,,NUL,os2 slibce,fillmode
   rc fillmode.res

fillmode.exe: fillmode.res
   rc fillmode.res
```

```
; ------------
; FILLMODE.DEF
; ------------

NAME        FILLMODE WINDOWAPI

DESCRIPTION 'Using different fill modes'

PROTMODE
STACKSIZE   4096
```

```
/* ----------- */
/* FILLMODE.RC */
/* ----------- */
```

```
#include <os2.h>
#include "fillmode.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "~Nofill", ID_NOFILL
      MENUITEM "~Alternate", ID_ALTERNATE
      MENUITEM "~Winding", ID_WINDING
      SUBMENU "~Rotation", ID_DIRECTIONU
         BEGIN
            MENUITEM "Coun~terCW", ID_COUNTERCW, , MIA_CHECKED
            MENUITEM "~Clockwise", ID_CLOCKWISE
         END
   END
```

Alternate Mode

Alternate mode is defined like this: an area should be filled if, in going—to the right—from a point in the area to infinity (far outside the picture), an odd number of boundaries is crossed. If an even number of boundaries is crossed, the area should not be filled.

Line Direction

Winding mode is more complex. To understand winding mode, you need to know that all lines and arcs are drawn with a certain *direction*. In figures drawn with GpiPolyLine, the direction of the line is specified by the sequence of points. For GpiBox the direction is determined by the order in which the points specify the corners, as we'll see in a moment. The direction of an arc depends on the arc parameters P, Q, R, and S. If P*Q is greater than R*S, the direction is counterclockwise; but if P*Q is less than R*S, it's clockwise. (P*Q equals R*S produces a straight line.) The default arc (a circle where P=Q=1 and R=S=0) is drawn counterclockwise.

Winding Mode

Winding mode is defined like this. To determine if a particular area in the figure should be filled, go—to the right—from a point in the area to infinity, and count +1 every time you cross an upward-going line, and −1 every time you cross a downward-going line. If you end up with a nonzero value, fill the area. If you end up with zero, don't fill it.

The FILLMODE program demonstrates two instances of winding mode: with the inner box drawn counterclockwise, and with it drawn clockwise. Figure 12-6 shows how the figure looks in these cases.

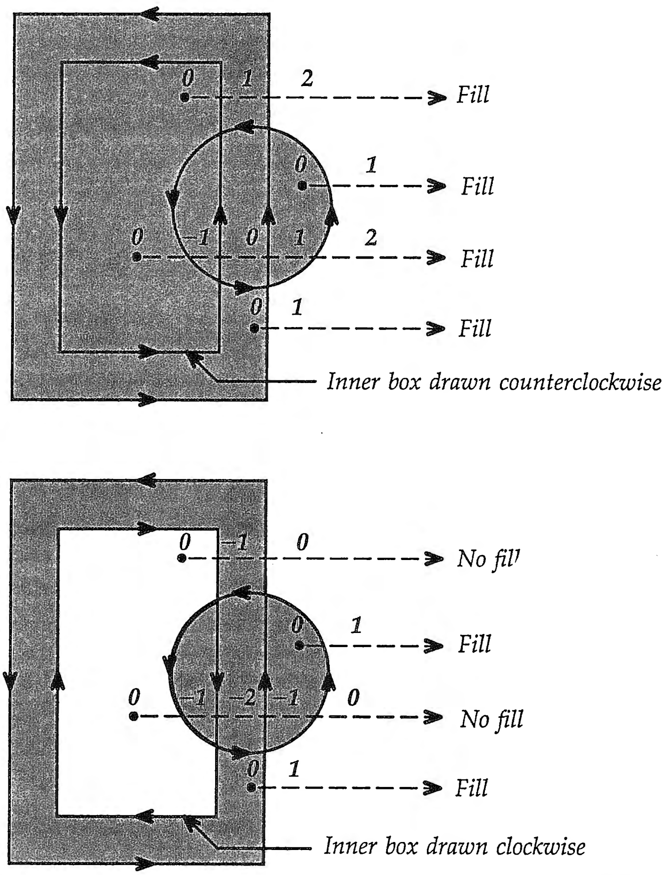


Figure 12-6. Winding mode

In the case of GpiBox the direction of the box's outline is determined by the two points specifying the corners of the box. Here's the rule: When you go from the current position to the second point, the X coordinate always changes first, not the Y. The other three lines then continue in the same direction. This is shown in Figure 12-7.

This means that if you specify the lower left and upper right corners, the box will be drawn counterclockwise. If you specify the lower right and upper left corners, the box will be drawn clockwise. We specify the points for the inner box differently in FILLMODE, depending on which direction the user has selected from the menu.

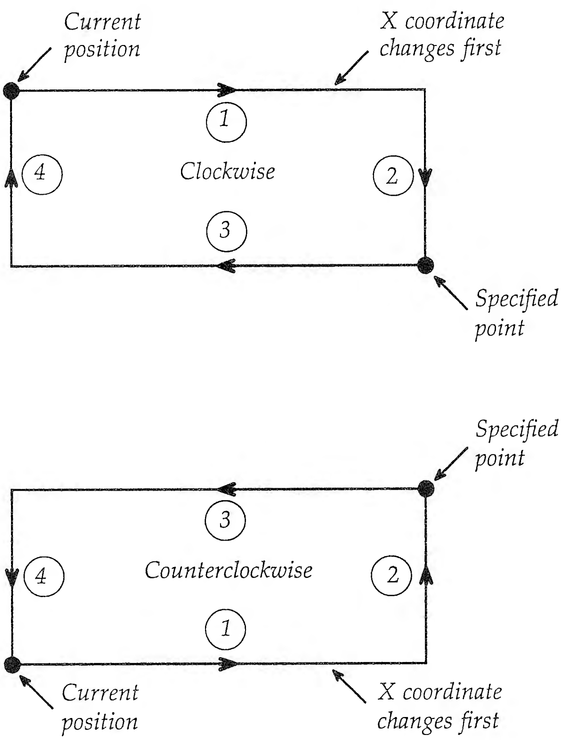


Figure 12-7. Line direction in GpiBox

To summarize the fill modes: Use alternate mode for doughnuts, and winding mode for airplane rivets, drawing the rivets in the same direction as the airplane. In more complicated situations you should be able to achieve any desired effect by selecting the fill mode and line direction appropriately.

PATHS

A path is defined as a series of lines and arcs. A path bracket is delimited by GpiBeginPath and GpiEndPath functions, in much the same way an area bracket is delimited by GpiBeginArea and GpiEndArea. Within a path bracket all the line-drawing APIs can be used: GpiLine, GpiPolyLine, GpiFullArc, and so on.

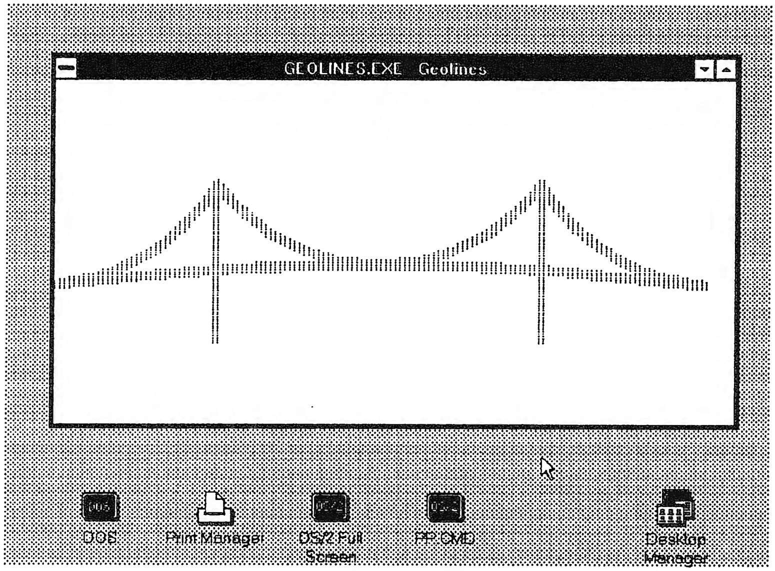


Figure 12-8. Output of the GEOLINES program

One use for paths is to draw lines of varying thickness. In the examples we've seen so far, lines and arcs have always been one pixel wide. Paths permit lines to have any thickness, and to be given a pattern and a color just as areas can. Wide lines created with paths in this way are called *geometric lines.*

Our example program demonstrates geometric lines by creating a sketch of the Golden Gate Bridge. The lines are drawn with a thickness of 0.12 inch, given a pattern, and colored dark red (as the bridge is in real life). Figure 12-8 shows the result.

In the example the path bracket is defined and the figure drawn on receipt of a WM_PAINT message. The PS units are set to low English, so a coordinate of 400, for example, means 4 inches.

Here are the listings for GEOLINES.C, GEOLINES.H, GEOLINES, and GEOLINES.DEF.

```
/* ------------------------------------------- */
/* GEOLINES.C - Lines with geometric thickness */
/* ------------------------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "geolines.h"

USHORT cdecl main(void)
```

```
    {
    HAB hab;                              /* handle for anchor block */
    HMQ hmq;                              /* handle for message queue */
    QMSG qmsg;                            /* message queue element */
    HWND hwnd, hwndClient;                /* handles for windows */

                                          /* create flags */
    ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                          FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

    static CHAR szClientClass[] = "Client Window";

    hab = WinInitialize(NULL);            /* initialize PM usage */
    hmq = WinCreateMsgQueue(hab, 0);      /* create message queue */

                                          /* register client window class */
    WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                          /* create standard window */
    hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
               szClientClass, " - Geolines", 0L, NULL, 0, &hwndClient);

                                          /* messages dispatch loop */
    while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
       WinDispatchMsg(hab, &qmsg);

    WinDestroyWindow(hwnd);               /* destroy frame window */
    WinDestroyMsgQueue(hmq);              /* destroy message queue */
    WinTerminate(hab);                    /* terminate PM usage */

    return 0;
    }

                                          /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
    {
    HPS hps;
    static SIZEL sizl = {0, 0};           /* for GpiSetPS */
    static LONG idPath = 1L;              /* fixed in this version of OS/2 */
    static POINTL ptlCable[] =
                 {{000,175}, {100,205}, {200,300},     /* left */
                  {400,200},                           /* center */
                  {600,300}, {700,205}, {800,175}};    /* right */
    static POINTL ptlDeck[] = {{400,200}, {0,175}};
    static POINTL ptlLeftBase = {200,100};
    static POINTL ptlRightBase = {600,100};

    switch (msg)
       {
       case WM_PAINT:
          hps = WinBeginPaint(hwnd, NULL, NULL);
          GpiSetPS(hps, &sizl, PU_LOENGLISH);
          GpiBeginPath(hps, idPath);

          GpiMove(hps, &ptlLeftBase);       /* left tower */
          GpiLine(hps, &ptlCable[2]);

          GpiMove(hps, &ptlRightBase);      /* right tower */
          GpiLine(hps, &ptlCable[4]);
```

```
      GpiMove(hps, &ptlCable[0]);
      GpiPointArc(hps, &ptlCable[1]);  /* left cable curve */
      GpiPointArc(hps, &ptlCable[3]);  /* center cable curve */
      GpiPointArc(hps, &ptlCable[5]);  /* right cable curve */
      GpiPointArc(hps, &ptlDeck[0]);   /* deck */

      GpiEndPath(hps);
                                       /* set attributes */
      GpiSetPattern(hps, PATSYM_DENSE7);
      GpiSetLineWidthGeom(hps, 12L);
      GpiSetLineJoin(hps, LINEJOIN_ROUND);
      GpiSetColor(hps, CLR_DARKRED);

      GpiStrokePath(hps, idPath, 0L);  /* draw path */

      WinEndPaint(hps);
      break;

   case WM_ERASEBACKGROUND:
      return TRUE;
      break;

   default:
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }

return NULL;                        /* NULL / FALSE */
}
```

```
/* ---------- */
/* GEOLINES.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define LINE_WIDTH 20L
```

```
# ------------------
# GEOLINES Make file
# ------------------

geolines.obj: geolines.c geolines.h
   cl -c -G2s -W3 -Zp geolines.c

geolines.exe: geolines.obj geolines.def
   link /NOD geolines,,NUL,os2 slibce,geolines
```

```
; ------------
; GEOLINES.DEF
; ------------
```

```
NAME        GEOLINES WINDOWAPI

DESCRIPTION 'Draw lines with a geometric thickness'

PROTMODE
STACKSIZE   4096
```

Defining the Path Bracket

The GpiBeginPath function signals the beginning of a path bracket.

Start a Path Bracket

```
BOOL GpiBeginPath(hps, idPath)
HPS hps             Handle to presentation space
LONG idPath         Path identifier (always 1 in current OS/2)

Returns: GPI_OK if successful, GPI_ERROR if error
```

Other than the PS handle, the only argument to GpiBeginPath is the path identifier. Only one path at a time can be defined in the current version of PM, so this argument is always set to 1.

Ending the path bracket requires GpiEndPath, which takes only the PS handle as an argument.

End a Path Bracket

```
BOOL GpiEndPath(hps)
HPS hps            Handle to presentation space

Returns: GPI_OK if successful, GPI_ERROR if error
```

Within the path bracket two straight lines, created with GpiLine, draw the two bridge towers. The three cable curves, and the bridge deck, are created with GpiPointArc. This function draws an arc from the current position through two other points.

Draw Arc Through Three Points

```
LONG GpiPointArc(hps, pptl);
HPS hps              Handle to presentation space
PPOINTL pptl         Array containing 2nd and 3rd points

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

The current arc parameters specify the shape of the arc. In this example the default is used, so the arcs are parts of circles. (Actually the bridge cable describes a catenary curve, but that's considerably harder to draw.)

The various points along the bridge cable and the deck are placed in arrays. Since GpiPointArc leaves the current position at the last of the three points, we can draw the series of arcs without using GpiMove to move the current position between arcs. Figure 12-9 shows how the points relate to the arrays.

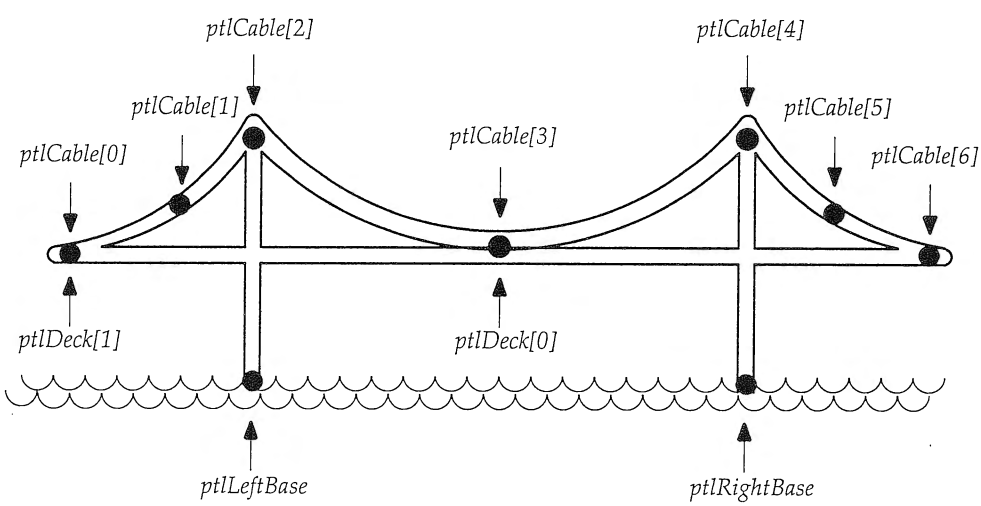


Figure 12-9. Points for GpiPointArc

Setting the Path Attributes

Various attributes can be set for the path, either before or after the path bracket is defined. In our example, GpiSetPattern selects the pattern to be used to fill the path, PATSYM_DENSE7. GpiSetColor selects the color, CLR_DARKRED.

A new API, GpiSetLineWidthGeom, specifies the width of the geometric lines.

Set Line Width

```
BOOL GpiSetLineWidthGeom(hps, lLineWidth)
HPS hps              Handle to presentation space
LONG lLineWidth      Line width

Returns: GPI_OK if successful, GPI_ERROR if error
```

The line width uses the FIXED data type, and is expressed in the units of the presentation space; in this case 0.01 inch.

The *join*—the point where two geometric lines meet—can be drawn in one of three ways: beveled, rounded, or mitred. In our example we round the ends of the joins using the GpiSetLineJoin API.

Set Line Join Attribute

```
BOOL GpiSetLineJoin(hps, flLineJoin)
HPS hps            Handle to presentation space
LONG flLineJoin    Line join type

Returns: GPI_OK if successful, GPI_ERROR if error
```

The *flLineJoin* argument can have one of the following values:

Identifier	Line Join Type
LINEJOIN_BEVEL	Bevel
LINEJOIN_DEFAULT	Default

LINEJOIN_MITRE	Mitre
LINEJOIN_ROUND	Round

Figure 12-10 shows how these joins look. The figure also shows different ways a geometric line can be ended. These can be specified with GpiSetLineEnd, although we don't use it in our example.

Stroking the Path

Once the path bracket is defined and the path attributes are set, the path can be drawn. This is carried out with the GpiStrokePath API. Drawing the edges of the geometric line is called *stroking* the path. GpiStrokePath first strokes the path, then fills in the space between the edges with the current color and fill pattern.

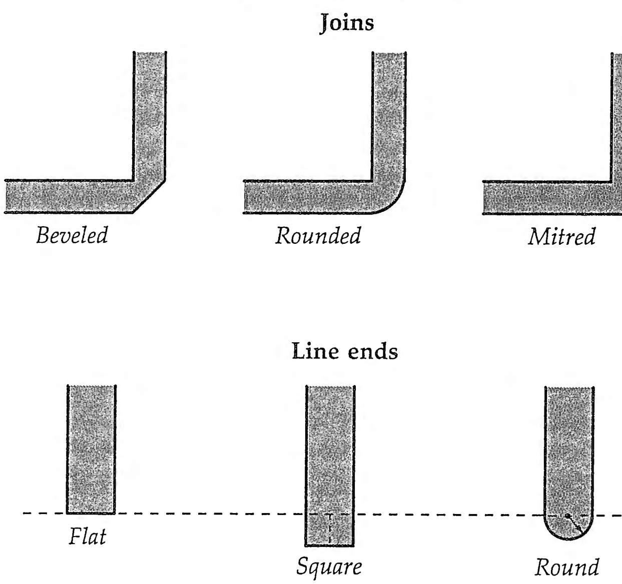


Figure 12-10. Line joins and ends

Stroke a Path (Draw a Geometric Line)

```
LONG GpiStrokePath(hps, lPath, flOptions)
HPS hps              Handle to presentation space
LONG lPath           Path to stroke (must be 1 in current OS/2)
ULONG flOptions      Reserved; must be 0

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

The *lPath* argument must be 1, and the *flOptions* argument must be 0 in the current version of OS/2.

Paths and Areas

Paths can also be used to fill the shape inside a closed figure formed from lines and arcs. This is called a *filled path* (as opposed to a stroked path) and is carried out with GpiFillPath. In this respect paths are very much like areas. We won't demonstrate filled paths.

REGIONS

A region consists of a rectangle, several rectangles that overlap each other, or several completely separate rectangles. PM can quickly make calculations involving the region and other graphics objects, so regions are useful in a variety of situations. For instance, you can conveniently determine whether a point lies within a region.

Another common use for regions is clipping: cropping off part of a graphics image. We'll explore this in the next section.

Regions can be combined in various ways to form new regions. They can also be moved, painted (made visible), and operated on in other ways.

In our example we'll use a region to determine whether a point lies in a certain rectangular area. The area is made visible on the screen by painting the region, using a fill pattern and color. Clicking the mouse inside this rectangle results in a beep, while clicking outside has no effect. The display is shown in Figure 12-11.

Here are the listings for REGION.C, REGION.H, REGION, and REGION.DEF.

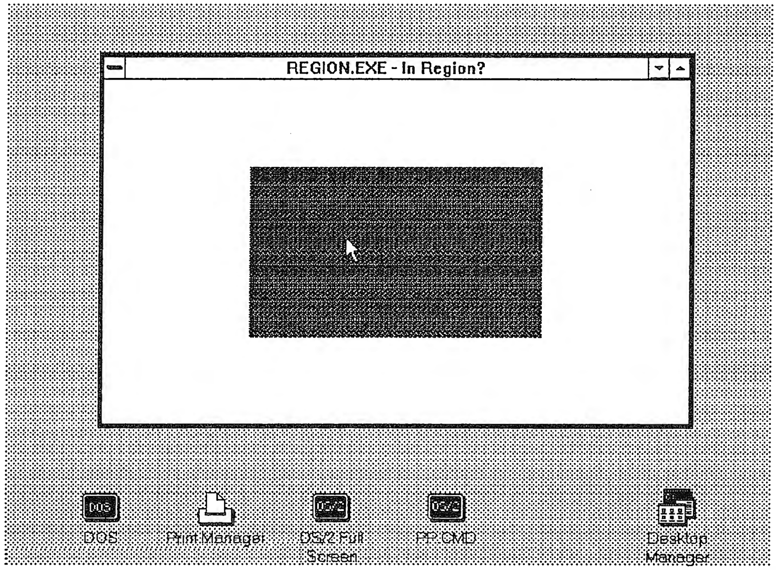


Figure 12-11. Output of the REGION program

```
/* ------------------------- */
/* REGION.C - Using a region */
/* ------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "region.h"

USHORT cdecl main(void)
   {
   HAB hab;                             /* handle for anchor block */
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd, hwndClient;               /* handles for windows */

                                        /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);     /* create message queue */

                                        /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                        /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
```

```
            szClientClass, " - In Region?", 0L, NULL, 0, &hwndClient);

                                        /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);              /* destroy frame window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */

   return 0;
   }

                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   HPS hps;
   static RECTL rcl;
   static HRGN hrgn;
   POINTL ptl;

   switch (msg)
      {
      case WM_BUTTON1DOWN:
         ptl.x = SHORT1FROMMP(mp1);
         ptl.y = SHORT2FROMMP(mp1);
         hps = WinGetPS(hwnd);
                                        /* check if in region */
         if (GpiPtInRegion(hps, hrgn, &ptl) == PRGN_INSIDE)
            WinAlarm(HWND_DESKTOP, WA_NOTE);
         WinReleasePS(hps);
         WinDefWindowProc(hwnd, msg, mp1, mp2);
         return TRUE;
         break;

      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
                                        /* create region */
         hrgn = GpiCreateRegion(hps, 1L, &rcl);
                                        /* paint region */
         GpiSetPattern(hps, PATSYM_HALFTONE);
         GpiSetBackColor(hps, CLR_GREEN);
         GpiPaintRegion(hps, hrgn);

         WinEndPaint(hps);
         break;

      case WM_SIZE:
         rcl.xLeft = SHORT1FROMMP(mp2) * .25;
         rcl.xRight = SHORT1FROMMP(mp2) * .75;
         rcl.yBottom = SHORT2FROMMP(mp2) * .25;
         rcl.yTop = SHORT2FROMMP(mp2) * .75;
         break;

      case WM_ERASEBACKGROUND:
        return TRUE;
        break;

      default:
```

```
        return(WinDefWindowProc(hwnd, msg, mp1, mp2));
        break;
    }

  return NULL;                          /* NULL / FALSE */
  }
```

```
/* -------- */
/* REGION.H */
/* -------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);
```

```
# ----------------
# REGION Make file
# ----------------

region.obj: region.c region.h
   cl -c -G2s -W3 -Zp region.c

region.exe: region.obj region.def
   link /NOD region,,NUL,os2 slibce,region
```

```
; ----------
; REGION.DEF
; ----------

NAME        REGION WINDOWAPI

DESCRIPTION 'Using a region'

PROTMODE
STACKSIZE   4096
```

Creating the Region

The region is created when the WM_PAINT message is received, using the GpiCreateRegion function.

Create a Region

```
HRGN GpiCreateRegion(hps, crcl, parcl)
HPS hps          Handle to presentation space
LONG crcl        Number of rectangles
PRECTL parcl     Array of rectangles of type RECTL

Returns: Handle to region if successful, 0 if error
```

GpiCreateRegion returns a region handle. This handle is used in other APIs that affect the region. There can be many regions in use at once.

A region can be created consisting of one or more rectangles. The *crcl* argument to GpiCreateRegion specifies the number of rectangles, and *parcl* is a pointer to an array of these rectangles. Each rectangle is defined by a structure of type RECTL.

In the example the region consists of a single rectangle. The coordinates of this rectangle are obtained during the processing of WM_SIZE. The program sizes the rectangle so it always occupies half the height and width of the window, and is centered in the window.

Painting the Region

Once the region has been defined, and any desired attributes such as color and pattern set, it can be painted. In the example we set the pattern to PATSYM_HALFTONE, and the color to CLR_GREEN. To paint the region we call GpiPaintRegion.

Paint (Fill) a Region

```
LONG GpiPaintRegion(hps, hrgn)
HPS hps     Handle to presentation space
HRGN hrgn   Handle to region

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

This function takes as arguments the handle to the region, in addition to the handle to the PS.

Is the Point in the Region?

To check if a specified point is in a certain region, we use the GpiPtIn-Region function.

Check Whether Point Is in Region

```
LONG GpiPtInRegion(hps, hrgn, pptl)
HPS hps             Handle to presentation space
HRGN hrgn           Handle to region
PPOINTL pptl        Point to be checked

Returns: PRGN_OUTSIDE if point outside, PRGN_INSIDE if inside,
         PRGN_ERROR if error
```

The arguments to this function specify the region and the point. The return value indicates if the point is inside the region. In the example we use WinAlarm to beep the speaker if the point is inside.

Destroying the Region

When a region is no longer needed, it could be destroyed with the GpiDestroyRegion function.

Destroy a Region

```
BOOL GpiDestroyRegion(hps, hrgn)
HPS hps             Handle to presentation space
HRGN hrgn           Handle to region

Returns: GPI_OK if successful, GPI_ERROR if error
```

BITMAPS

The Gpi drawing functions in previous examples created *vector graphics.* These are graphics objects, like lines, that are specified by a few control points. It is also possible to create a graphics object by specifying the pixels from which it is drawn. This is called a *bitmapped* graphics object.

Bitmapped graphics are useful in a variety of situations. They can create graphics objects that are impractical or impossible to create with vector graphics, such as small, highly detailed pictures. Icons and mouse pointers, which we discussed in Chapter 7, are examples of bitmaps. Bitmaps can be used for custom fill patterns. They are also used in more complex endeavors, such as manipulating video pictures (which are bitmap or *raster* based), and pictures captured with a scanner.

A *bitmap* is the data—bits and bytes—that represents the graphic object. In a monochrome bitmap each bit in the bitmap represents one pixel in the object. The first bit in the bitmap corresponds to the lower left corner of the object, as shown in Figure 12-12.

Bitmaps must be rectangular, but the number of pixels in the horizontal and vertical directions can vary. Of course, the larger the bitmap, the longer it takes to load it, move it, or perform other operations on it.

Colored bitmaps are similar, but use more than one bit to represent each pixel in the object. The number of bits used depends on the number of

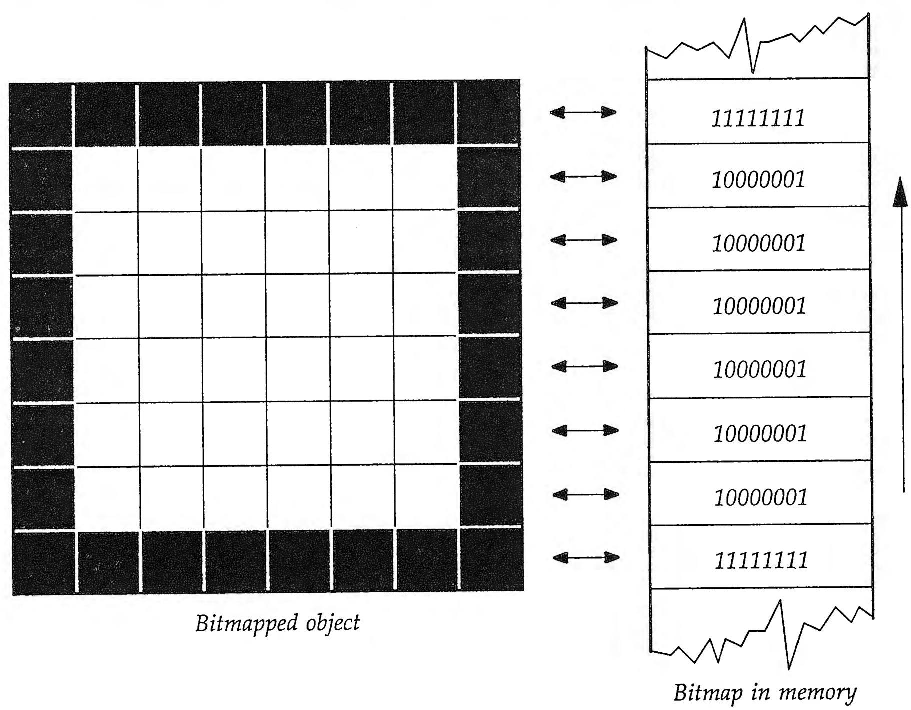


Figure 12-12. A bitmap and a bitmapped object

colors. This is called the *bitcount*. In PM it can be 1, 4, 8, or 24, which provide 2, 16, 256, or 16,777,216 colors, respectively.

Vector graphics are device independent. A circle drawn with GpiFullArc (assuming the presentation units are not pixels) will look roughly the same on all devices, whether screen, printer, or plotter. A bitmap, on the other hand, will look different on different devices. It will be much smaller on a 300 dpi laser printer than on the screen because there are more pixels per inch on the printer. Since the aspect ratio of pixels is different on different devices, bitmaps will also have different proportions on different devices. Bitmaps cannot even be drawn on purely vector devices, such as most plotters.

Creating Bitmaps

A bitmap can be created in several ways. One approach is to create an array in memory in which each bit (or group of bits, if color is used) represents one pixel. Here's an array representing a monochrome bitmap that generates the outline of a square, 8 pixels on a side, as seen in the earlier figure:

```
static BYTE abSquare[] =
          {0xFF, 0x81, 0x81, 0x81, 0x81, 0x81, 0x81, 0xFF};
```

This approach becomes tedious for any but the simplest bitmaps, since the picture must be translated into hex digits by hand. It's far easier to use a special-purpose editor to create the bitmap. The same editor, ICONEDIT, used to create icons and pointers, is also used to create bitmaps.

Bitmaps as Patterns

Bitmaps are commonly used to create custom fill patterns. These can be used to supplement the standard patterns. For example, you can create patterns representing brick, shingle, and stone for depicting buildings, and patterns representing land forms like swamps and forests for drawing maps.

Here's how to set up a custom pattern:

- Create an 8 by 8 bitmap pattern with the icon editor.
- Associate the resulting .BMP file with an ID, using a BITMAP statement in the .RC file.
- Load the bitmap with GpiLoadBitmap.

- Associate the bitmap with a tag, using GpiSetBitmapId.
- Set the pattern to this tag using GpiSetPatternSet.
- Draw an area; it will be filled with the custom pattern.

Our example program uses a custom pattern to fill a box. Here are the listings for CUSTPAT.C, CUSTPAT.H, CUSTPAT, CUSTPAT.DEF, and CUSTPAT.RC.

```
/* ---------------------------------- */
/* CUSTPAT.C - Create a custom pattern */
/* ---------------------------------- */

#define INCL_WIN
#define INCL_GPI
#include <os2.h>

#include "custpat.h"

USHORT cdecl main(void)
   {
   HAB hab;                            /* handle for anchor block */
   HMQ hmq;                            /* handle for message queue */
   QMSG qmsg;                          /* message queue element */
   HWND hwnd, hwndClient;              /* handles for windows */

                                       /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);          /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);    /* create message queue */

                                       /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                       /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Custom Pattern",
             0L, NULL, ID_FRAMERC,  &hwndClient);

                                       /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);             /* destroy frame window */
   WinDestroyMsgQueue(hmq);            /* destroy message queue */
   WinTerminate(hab);                  /* terminate PM usage */

   return 0;
   }

                                       /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
```

```
                   MPARAM mp2)
{
HPS hps;                                /* PS handle */
HBITMAP hbm;                            /* bitmap handle */
static POINTL ptlBoxLL;                 /* box lower left corner */
static POINTL ptlBoxUR;                 /* box upper right corner */

switch (msg)
   {
   case WM_PAINT:
      hps = WinBeginPaint(hwnd, NULL, NULL);
      GpiErase(hps);
                                        /* load bitmap */
      hbm = GpiLoadBitmap(hps, NULL, ID_PATTERNBMP, 0L, 0L);
                                        /* tag bitmap */
      GpiSetBitmapId(hps, hbm, BITMAP_TAG);
                                        /* set pattern to tag */
      GpiSetPatternSet(hps, BITMAP_TAG);
                                        /* draw box */
      GpiMove(hps, &ptlBoxLL);          /* with pattern */
      GpiBox(hps, DRO_OUTLINEFILL, &ptlBoxUR, 0L, 0L);

      GpiDeleteSetId(hps, BITMAP_TAG);  /* remove tag */
      GpiDeleteBitmap(hbm);             /* delete bitmap */
      WinEndPaint(hps);
      break;

   case WM_SIZE:                        /* size the box */
      ptlBoxLL.x = SHORT1FROMMP(mp2) * 0.25;
      ptlBoxLL.y = SHORT2FROMMP(mp2) * 0.25;
      ptlBoxUR.x = SHORT1FROMMP(mp2) * 0.75;
      ptlBoxUR.y = SHORT2FROMMP(mp2) * 0.75;
      break;

   default:
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }

return NULL;                        /* NULL / FALSE */
}
```

```
/* --------- */
/* CUSTPAT.H */
/* --------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1

#define ID_PATTERNBMP 100
#define BITMAP_TAG 200L
```

```
# -----------------
# CUSTPAT Make file
# -----------------

custpat.obj: custpat.c custpat.h
```

```
   cl -c -G2s -W3 -Zp custpat.c

custpat.res: custpat.rc custpat.h basket.bmp
   rc -r custpat.rc

custpat.exe: custpat.obj custpat.def
   link /NOD custpat,,NUL,os2 slibce,custpat
   rc custpat.res custpat.exe

custpat.exe: custpat.res
   rc custpat.res
```

```
; -----------
; CUSTPAT.DEF
; -----------

NAME        CUSTPAT    WINDOWAPI

DESCRIPTION 'Create a custom pattern'

PROTMODE
STACKSIZE   4096
```

```
/* ---------- */
/* CUSTPAT.RC */
/* ---------- */

#include <os2.h>
#include "custpat.h"

BITMAP ID_PATTERNBMP basket.bmp
```

The pattern is created using the icon editor on an 8x8 pixel grid. Figure 12-13 shows the bitmap for the pattern, and Figure 12-14 shows the display when the program is executed. The pattern is reminiscent of a woven basket.

Each bitmap file must be associated with a unique ID number. This is done in the .RC file, using the BITMAP keyword.

```
BITMAP ID_PATTERNBMP basket.bmp
```

Loading a Bitmap

The bitmap is loaded from the resource file into the presentation space using the GpiLoadBitmap function.

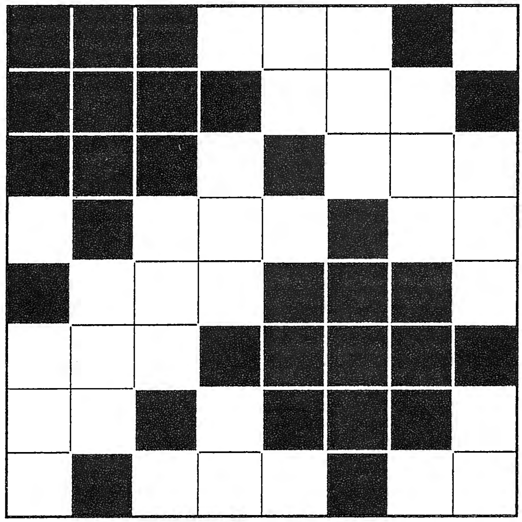


Figure 12-13. Custom pattern

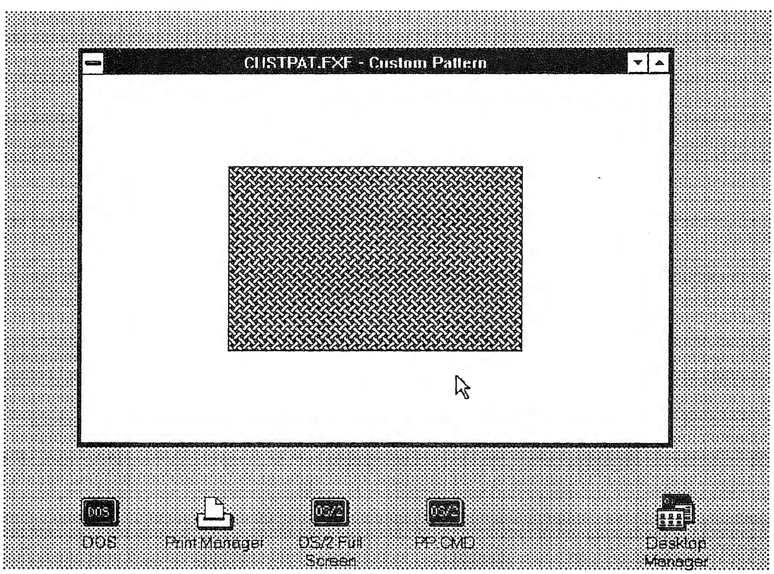


Figure 12-14. Output of the CUSTPAT program

Load Bitmap Resource, Convert to Bitmap

```
HBITMAP GpiLoadBitmap(hps, hmod, idBitmap, lWidth, lHeight)
HPS hps            Handle to presentation space
HMODULE hmod       Module handle (NULL if in .EXE file)
USHORT idBitmap    Bitmap identifier in resource file
LONG lWidth        Bitmap width (pixels)
LONG lHeight       Bitmap height (pixels)

Returns: Bitmap handle if successful, GPI_ERROR if error
```

If either the *lWidth* or *lHeight* argument is zero, the dimensions of the bitmap are assumed to be those specified in the bitmap resource. That is, the bitmap is loaded unchanged. However, if values other than 0 are used, the bitmap will be stretched or compressed to fit the new width and height. In our example the bitmaps are loaded unchanged.

Tagging a Bitmap

To be used as a pattern, a bitmap must be given a local identifier or *tag*. The GpiSetBitmapId function performs this service.

Tag Bitmap with Local Identifier

```
BOOL GpiSetBitmapId(hps, hbm, lcid)
HPS hps             Handle to presentation space
HBITMAP hbm         Handle to bitmap
LONG lcid           Local identifier

Returns: GPI_OK if successful, GPI_ERROR if error
```

The tag is used in the GpiSetPatternSet function to set the current pattern set to the custom pattern. The idea of a pattern set applies more to fonts than to bitmaps, but the same API is used for both.

Setting a Pattern Set

Set Current Pattern Set Attribute

```
BOOL GpiSetPatternSet(hps, lcid)
HPS hps              Handle to presentation space
LONG lcid            Local identifier for font or bitmap

Returns: GPI_OK if successful, GPI_ERROR if error
```

The pattern set can be a tagged bitmap, as shown in the example, or it can be a font. If it's a font, the GpiSetPatternSet function selects a font, and GpiSetPattern selects a character from the font. This character is then used for the pattern. We won't explore this possibility.

Once GpiSetPatternSet has been executed, graphics areas can be drawn in the usual way, either with GpiBox or GpiFullArc, or as a graphics bracket, and they will automatically be filled with the custom pattern. Each 0 in the pattern will cause the corresponding pixel to be drawn in the background color, while a 1 in the pattern will produce a pixel in the foreground color. In our example we draw a box with GpiBox. The box is sized during WM_SIZE so that it always occupies the same proportion of the window.

Removing the Tag

When the custom pattern is no longer needed, the tag is removed with the GpiDeleteSetId function. This function does not delete the bitmap itself; its handle remains available for use. To delete the bitmap requires another function, GpiDeleteBitmap.

Delete Bitmap

```
BOOL GpiDeleteBitmap(hbm)
HBITMAP hbm         Handle to bitmap

Returns: GPI_OK if successful, GPI_ERROR if error
```

Bitmaps as Graphics Objects

In our next example, four bitmaps depict the suits in a deck of cards: hearts, diamonds, spades, and clubs. The user selects a suit from the menu, and the program then draws a card of that suit on the screen. For simplicity the card is always an ace; only the suit varies. Figure 12-15 shows the output when spades are selected.

In this program the suit and number in the lower right corner are not drawn; we'll rectify this omission in the next example.

The first step in the program is creating the bitmaps. To simplify the programming, all the suits are drawn in a grid 19 pixels wide and 30 pixels high, although some suits don't use the full height. Figure 12-16 shows what the spade bitmap looks like. The other suits are created similarly.

The bitmaps were saved in files called HEART.BMP, DIAMOND.BMP, and so on. Another bitmap, for the letter 'A' for Ace, was created and saved in a file called A.BMP.

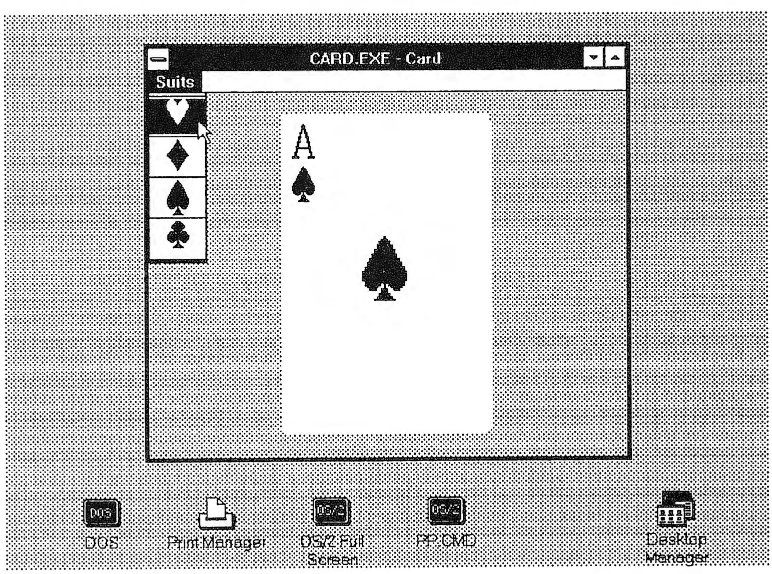


Figure 12-15. Output of the CARD program

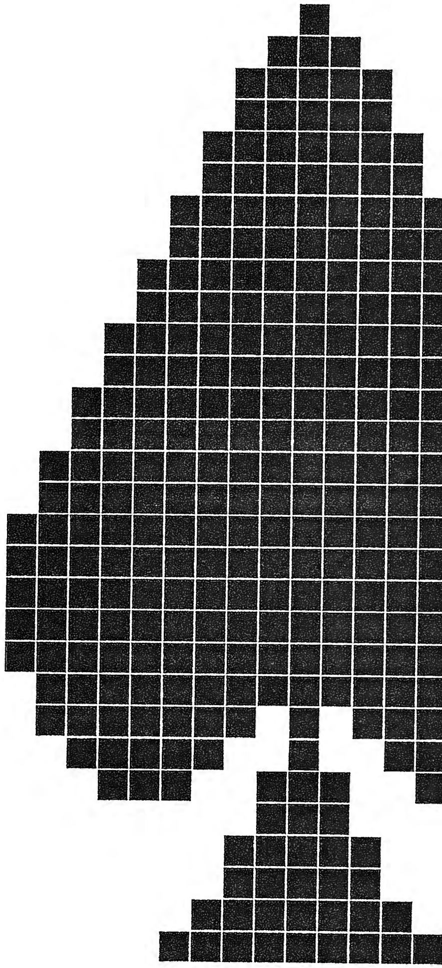


Figure 12-16. The spade bitmap

Here are the listings for CARD.C, CARD.H, CARD, CARD.DEF, and CARD.RC.

```
/* -------------------- */
/* CARD.C - Draw a card */
/* -------------------- */

#define INCL_WIN
#include <os2.h>

#include "card.h"
```

```
USHORT cdecl main(void)
   {
   HAB hab;                                /* handle for anchor block */
   HMQ hmq;                                /* handle for message queue */
   QMSG qmsg;                              /* message queue element */
   HWND hwnd, hwndClient;                  /* handles for windows */

                                           /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_MENU |
                         FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);              /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);        /* create message queue */

                                           /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                           /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
           szClientClass, " - Card", 0L, NULL, ID_FRAMERC,  &hwndClient);

                                           /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);                 /* destroy frame window */
   WinDestroyMsgQueue(hmq);                /* destroy message queue */
   WinTerminate(hab);                      /* terminate PM usage */

   return 0;
   }
                                           /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                     MPARAM mp2)
   {
   HPS hps;
   HBITMAP hbmSuit, hbmNumber;
   RECTL rcl;
   static USHORT idBmp = 0;                /* ID number of bitmap */
   static POINTL ptlSuit = {120, 220};     /* suit in UL corner of card */
   static POINTL ptlNumber = {120, 257};   /* number in UL corner of card */
   static RECTL rclCenter =                /* suit in center of card */
               {178, 134, 216, 190};
   static POINTL ptlBoxLL = {110, 20};     /* LL corner of card */
   static POINTL ptlBoxUR = {285, 295};    /* UR corner of card */
   static LONG clrFg;                      /* foreground color of bitmap */

   switch (msg)
      {
      case WM_COMMAND:
         {
         switch (COMMANDMSG(&msg) -> cmd)
            {
                                           /* suit change */
            case ID_HEARTBMP:
            case ID_DIAMONDBMP:
            case ID_SPADEBMP:
```

```
            case ID_CLUBBMP:
               idBmp = COMMANDMSG(&msg) -> cmd;  /* match color to suit */
               if (idBmp == ID_HEARTBMP || idBmp == ID_DIAMONDBMP)
                  clrFg = CLR_RED;
               else
                  clrFg = CLR_BLACK;

               WinSetWindowPos(WinQueryWindow(hwnd, QW_PARENT, FALSE),
                  NULL, 100, 5, 395, 345, SWP_MOVE | SWP_SIZE);
               WinInvalidateRect(hwnd, NULL, FALSE);
               break;
            }
         break;
         }
      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
         WinQueryWindowRect(hwnd, &rcl);      /* color background */
         WinFillRect(hps, &rcl, CLR_DARKGREEN);
         if (idBmp != 0)
            {
            GpiSetColor(hps, CLR_WHITE);      /* draw card */
            GpiMove(hps, &ptlBoxLL);          /* with rounded corners */
            GpiBox(hps, DRO_OUTLINEFILL, &ptlBoxUR, 15L, 15L);

                                              /* load bitmaps */
            hbmSuit = GpiLoadBitmap(hps, NULL, idBmp, 0L, 0L);
            hbmNumber = GpiLoadBitmap(hps, NULL, ID_ABMP, 0L, 0L);

                                              /* draw corner suit */
            WinDrawBitmap(hps, hbmSuit, NULL, &ptlSuit, clrFg,
                          CLR_WHITE, DBM_NORMAL);
                                              /* draw corner number */
            WinDrawBitmap(hps, hbmNumber, NULL, &ptlNumber, clrFg,
                          CLR_WHITE, DBM_NORMAL);
                                              /* draw center suit */
            WinDrawBitmap(hps, hbmSuit, NULL, (PPOINTL)&rclCenter, clrFg,
                          CLR_WHITE, DBM_STRETCH);

            GpiDeleteBitmap(hbmNumber);       /* delete bitmaps */
            GpiDeleteBitmap(hbmSuit);
            }
         WinEndPaint(hps);
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                       /* NULL / FALSE */
   }
```

```
/* ------ */
/* CARD.H */
/* ------ */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);
```

```
#define ID_FRAMERC 1

#define ID_HEARTBMP 20
#define ID_DIAMONDBMP 30
#define ID_SPADEBMP 40
#define ID_CLUBBMP 50

#define ID_ABMP 100
```

```
# --------------
# CARD Make file
# --------------

card.obj: card.c card.h
   cl -c -G2s -W3 -Zp card.c

card.res: card.rc card.h
   rc -r card.rc

card.exe: card.obj card.def
   link /NOD card,,NUL,os2 slibce,card
   rc card.res card.exe

card.exe: card.res
   rc card.res
```

```
; --------
; CARD.DEF
; --------

NAME        CARD    WINDOWAPI

DESCRIPTION 'Draw a card'

PROTMODE
STACKSIZE   4096
```

```
/* ------- */
/* CARD.RC */
/* ------- */

#include <os2.h>
#include "card.h"

BITMAP ID_HEARTBMP heart.bmp
BITMAP ID_DIAMONDBMP diamond.bmp
BITMAP ID_SPADEBMP spade.bmp
BITMAP ID_CLUBBMP club.bmp
BITMAP ID_ABMP a.bmp

MENU ID_FRAMERC
   BEGIN
      SUBMENU "Suits", 100
         BEGIN
```

```
            MENUITEM "", 101, MIS_SEPARATOR
            MENUITEM "#20", ID_HEARTBMP, MIS_BITMAP
            MENUITEM "",  102, MIS_SEPARATOR
            MENUITEM "#30", ID_DIAMONDBMP, MIS_BITMAP
            MENUITEM "", 103, MIS_SEPARATOR
            MENUITEM "#40", ID_SPADEBMP, MIS_BITMAP
            MENUITEM "", 104, MIS_SEPARATOR
            MENUITEM "#50", ID_CLUBBMP, MIS_BITMAP
            MENUITEM "", 105, MIS_SEPARATOR
        END
END
```

The suit bitmaps are used in two different ways in the program: as menu items, and as graphics elements in the client window.

Setting Bitmaps as Menu Items

It's surprisingly easy to use a bitmap as a menu item. You simply place the ID of the bitmap and the identifer MIS_BITMAP in a MENUITEM statement in the .RC file. For example,

```
MENUITEM "" ID_HEARTBMP, MIS_BITMAP
```

The bitmap is drawn instead of the menu item title string, which isn't used and can be empty. MENUITEM statements with MIS_SEPARATOR can be used to draw lines between the bitmapped items.

Drawing Bitmaps

Two steps are necessary to draw a bitmap in the client window. First the bitmap resource is loaded into the presentation space, using GpiLoadBitmap, as described earlier. Then the bitmap is drawn on the screen using WinDrawBitmap.

Draw Bitmap

```
BOOL WinDrawBitmap(hpsDst, hbm, prclSrc, pptlDst, clrFore,
                   clrBack, fs)
HBS hpsDst         Presentation space in which bitmap will be drawn
HBITMAP hbm        Bitmap handle
PRECTL prclSrc     Coordinates of bitmap (NULL for entire bitmap)
PPOINTL pptlDst    Point at lower left corner of destination
LONG clrFore       Bitmap foreground color
LONG clrBack       Bitmap background color
USHORT fs          Bitmap flags

Returns: TRUE if successful, FALSE if error
```

The *hpsDst* argument is the PS where the bitmap will be drawn, and *hbm* is the bitmap handle. The *prclSrc* argument is the lower left corner of the bitmap. The *clrFore* and *clrBack* arguments specify the bitmap's foreground and background colors.

The *fs* argument can have the following values:

Identifier	**Bitmap is Drawn with**
DBM_HALFTONE	Alternating ones and zeros pattern
DBM_IMAGEATTRS	Colors of *hpsDst; clrFore* and *clrBack* ignored
DBM_INVERT	Bitmap color inverted
DBM_NORMAL	Bitmap color normal
DBM_STRETCH	Stretched to fit RECTL structure in *pptlDst*

In our example the foreground color, *clrFore,* is set according to the menu selection: hearts and diamonds are red, and spades and clubs are black. The background color is set to CLR_WHITE, the color of the card. DBM_NORMAL, used to draw the bitmaps in the corner of the card, causes the colors to be displayed as is, and the bitmaps to be drawn normal size. The DBM_STRETCH identifier is used to draw the enlarged suit bitmap in the center of the card. This identifier causes the *pptlDst* argument to be interpreted as a rectangle instead of a point. The rectangle holds the coordinates the bitmap will be enlarged (or condensed) to fit.

In this example we use constants for the coordinates of the card and bitmaps, but it would be easy to substitute variables, to facilitate moving the card.

Bit Blitting

We can change the size and shape of a bitmap with the GpiLoadBitmap and WinDrawBitmap functions, but we can't rotate it. In the CARD example, the suit and number in the lower right corner of the card should be upside down. To achieve this effect, we'll call into play a powerful graphics function: GpiBitBlt.

The GpiBitBlt Function

The purpose of GpiBitBlt is to move bits, or more specifically, a rectangular area of bits: a bitmap. *Bitblt* (pronounced "bit blit") means "bit block transfer." GpiBitBlt can move a bitmap as is, it can expand or compress it,

and it can flip it horizontally or vertically. It can move the bitmap from one presentation space to another. The source bits can be combined with the target bits in a variety of ways (similar to the way colors are combined using different mix modes), and a pattern may also be combined with the result in various ways.

Let's look at the GpiBitBlt function in detail before going on to the example.

Copy a Bitmap

```
LONG GpiBitBlt(hpsTarg, hpsSrc, cPoints, paptl, lRop, flOptions)
HPS hpsTarg          Target presentation space
HPS hpsSrc           Source presentation space
LONG cPoints         Number of points in paptl: 2, 3, or 4
PPOINTL paptl        LL target, UR target, LL source, UR source
LONG lRop            Operation: ROP_SRCCOPY, etc.
ULONG flOptions      Compression option: BBO_AND, etc.

Returns: GPI_OK if successful, GPI_ERROR if error
```

The *hpsTarg* and *hpsSrc* arguments are the target and source presentation spaces. Both presentation spaces must be associated with device contexts that support bitmaps: the screen display, a memory device context (more on this later), or a printer or other raster device that handles bitmaps; most plotters aren't appropriate. The target and source presentation spaces can be the same or different.

The coordinates of the source and/or the target bitmaps are specified in the *paptl* argument, an array of points. The number of points in this array is given by *cPoints.* This number determines how the points in *paptl* are interpreted. Here are the possibilities:

- If *cPoints* is 2, the two points in *paptl* are the lower left and upper right corners of the target rectangle. No source is used, only a pattern and the target. (This is not as common a situation as the next two.)
- If *cPoints* is 3, the three points in *paptl* are the lower left and upper right corners of the target, and the lower left corner of the source. The source is copied to the target with no size or orientation changes.
- If *cPoints* is 4, the four points in *paptl* are the lower left and upper right corners of the target, and the lower left and upper right corners of the

source. The size and orientation of the bitmap can be changed during the move.

The *lRop* argument specifies the way the bits in the source bitmap will be combined with the bits in the target, and optionally with a pattern. For instance, the source could be copied to the target only where bits were not set in the pattern. If the pattern represented bars, only the scene between the bars would be copied.

We saw with mix modes that there were 16 possible ways to combine a monochrome object with a monochrome drawing surface. When a pattern is used as well, there are 256 possible combinations. The object can be 0 or 1, as can the drawing surface and the pattern. There are eight combinations of these three items, 000 through 111 binary. A mix mode specifies what each of these eight combinations yields: 000=1, 001=0, 010=1, and so on. This requires an eight-digit binary number, like 10101100. Eight binary digits (two hex digits) can represent 256 mix modes, so this is the maximum. All of these can be used, but only a few are given identifiers. These are

Identifier	**To Derive Target**
ROP_DSTINVERT	Invert target
ROP_MERGECOPY	AND source and pattern
ROP_MERGEPAINT	OR inverse source and target
ROP_NOTSRCCOPY	Invert source
ROP_NOTSRCERASE	AND inverse source and inverse target
ROP_ONE	Set all bits to 1
ROP_PATCOPY	Copy pattern
ROP_PATINVERT	XOR pattern and target
ROP_PATPAINT	OR inverse source, pattern, and target
ROP_SRCAND	AND source and target
ROP_SRCCOPY	Copy source to target
ROP_SRCERASE	AND source and inverse target
ROP_SRCINVERT	XOR source and target
ROP_SRCPAINT	OR source and target
ROP_ZERO	Set all bits to 0

A common choice here is ROP_SRCCOPY, which replaces all the pixels in the target with pixels from the source.

The *flOptions* argument applies when the target is smaller than the source. It specifies one of three algorithms to use in eliminating the unneeded rows or columns of bits. The possibilities are

Identifier	To Compress Two Rows or Columns
BBO_AND	AND them together
BBO_IGNORE	Delete one
BBO_OR	OR them together (default)

BBO_AND is useful for compressing black images on a white background, BBO_OR for white images on a black background, and BBO_IGNORE is useful for color on color.

Examples of GpiBitBlt

To copy a bitmap from one place in a window to another, you would use the same presentation space for both source and target. Assuming the size and orientation of the bitmap are not to be changed, and a 25 by 25 pixel image is to be moved from (100, 100) to (200, 300), the code might look like this:

```
POINTL apl[3] = {{200, 300}, {225, 325}, {100, 100}};
  .
  .
  .
GpiBitBlt(hps, hps, 3L, aptl, ROP_SRCCOPY, 0L);
```

If the bitmap were to be expanded to twice its previous height and width, the code would be

```
POINTL apl[4] = {{200, 300}, {250, 350}, {100, 100}, {125, 125}};
  .
  .
  .
GpiBitBlt(hps, hps, 4L, aptl, ROP_SRCCOPY, 0L);
```

To flip the bitmap, change the order of the target coordinates. For an upside-down flip, interchange the Y coordinates:

```
POINTL apl[4] = {{200, 350}, {250, 300}, {100, 100}, {125, 125}};
```

For a left-right flip, interchange the X coordinates:

```
POINTL apl[4] = {{250, 300}, {200, 350}, {100, 100}, {125, 125}};
```

The Memory Device Context

As we noted, the source and destination presentation spaces used in GpiBitBlt can be different. You could copy a bitmap from the display to the printer, for example, by associating the source PS with the display, and the target PS with the printer. Probably the most common use of GpiBitBlt, however, is copying to and from a *memory device context.*

A memory device context is simply a section of memory that acts like a device. Bitmaps can be drawn to it, and read back again. It provides a convenient place to store bitmaps until they are needed.

The memory device context is created using the OD_MEMORY option in DevOpenDC. Then a presentation space is opened and associated with this device context:

```
hdcMemory = DevOpenDC(hab, OD_MEMORY, "*", 0L, NULL, NULL);
hpsMemory = GpiCreatePS(hab, hdcMemory, &sizl,
                PU_PELS | GPIT_MICRO | GPIA_ASSOC);
```

The CARDBLT Example

The next example is similar to CARD, but also draws the suit and number, upside down, in the lower right corner of the card, as shown in Figure 12-17.

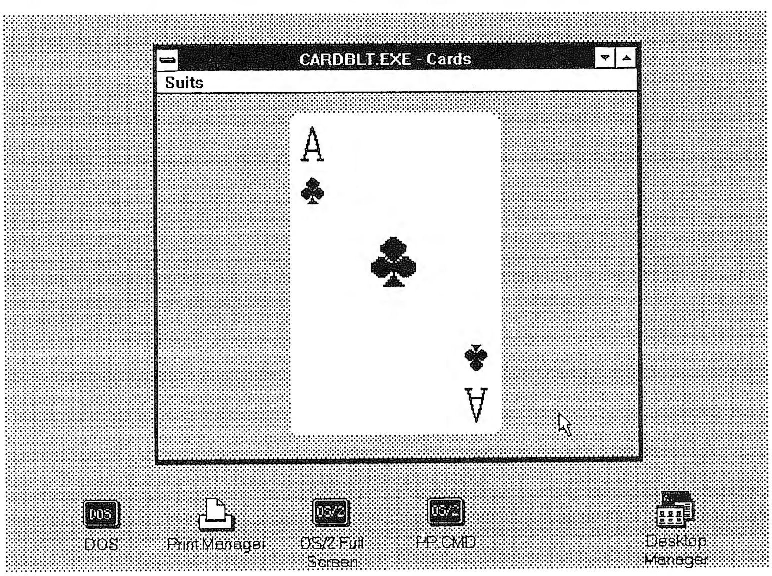


Figure 12-17. Output of the CARDBLT program

Since GpiBitBlt is used to draw the upside-down bitmaps, we draw the other bitmaps the same way, instead of with WinDrawBitmap. A memory device context is used to store the bitmaps after they're loaded.

Here are the listings for CARDBLT.C, CARDBLT.H, CARDBLT, CARDBLT.DEF, and CARDBLT.RC.

```
/* ----------------------------------- */
/* CARDBLT.C - Draw a card with bitblt */
/* ----------------------------------- */

#define INCL_WIN
#include <os2.h>
#include "cardblt.h"

HAB hab;                                /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd, hwndClient;               /* handles for windows */

                                        /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_MENU |
                         FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);     /* create message queue */

                                        /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                        /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Cards", 0L, NULL, ID_FRAMERC, &hwndClient);

                                        /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);              /* destroy frame window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */

   return 0;
   }

                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   static HDC hdcMemory;
   static HPS hpsMemory;
   HPS hps;
```

```
HBITMAP hbmSuit, hbmNumber;
RECTL rcl;
static USHORT idBmp = 0;
static POINTL ptlBoxLL = {110, 20};
static POINTL ptlBoxUR = {285, 295};
static LONG clrFg;
POINTL aptl[4];
static SIZEL sizl = {0,0};

switch (msg)
   {
   case WM_CREATE:
                                          /* open memory DC */
      hdcMemory = DevOpenDC(hab, OD_MEMORY, "*", 0L, NULL, NULL);
                                          /* open memory PS */
      hpsMemory = GpiCreatePS(hab, hdcMemory, &sizl,
                     PU_PELS | GPIT_MICRO | GPIA_ASSOC);
      break;

   case WM_COMMAND:
      {
      switch (COMMANDMSG(&msg) -> cmd)
         {
                                          /* suit change */
         case ID_HEARTBMP:
         case ID_DIAMONDBMP:
         case ID_SPADEBMP:
         case ID_CLUBBMP:
            idBmp = COMMANDMSG(&msg) -> cmd;
            if (idBmp == ID_HEARTBMP || idBmp == ID_DIAMONDBMP)
               clrFg = CLR_RED;
            else
               clrFg = CLR_BLACK;
            WinSetWindowPos(WinQueryWindow(hwnd, QW_PARENT, FALSE),
               NULL, 100, 5, 395, 345, SWP_MOVE | SWP_SIZE);
            WinInvalidateRect(hwnd, NULL, FALSE);
            break;
         }
      break;
      }
   case WM_PAINT:
      hps = WinBeginPaint(hwnd, NULL, NULL);
      WinQueryWindowRect(hwnd, &rcl);       /* color background */
      WinFillRect(hps, &rcl, CLR_DARKGREEN);
      if (idBmp != 0)
         {
         GpiSetColor(hps, CLR_WHITE);       /* draw card */
         GpiMove(hps, &ptlBoxLL);           /* with rounded corners */
         GpiBox(hps, DRO_OUTLINEFILL, &ptlBoxUR, 15L, 15L);

         GpiSetColor(hps, clrFg);           /* set bitmap color */

                                            /* load suit bitmap */
         hbmSuit = GpiLoadBitmap(hpsMemory, NULL, idBmp, 0L, 0L);

         GpiSetBitmap(hpsMemory, hbmSuit);  /* set suit bitmap */

                                            /* draw UL corner suit */
         aptl[0].x = 120; aptl[0].y = 211;    /* LL target */
         aptl[1].x = 139; aptl[1].y = 241;    /* UR target */
```

```
            aptl[2].x =   0; aptl[2].y =   0;   /* LL source */
            GpiBitBlt(hps, hpsMemory, 3L, aptl, ROP_SRCCOPY, 0L);

                                               /* draw center suit */
            aptl[0].x = 178; aptl[0].y = 134;   /* LL target */
            aptl[1].x = 216; aptl[1].y = 190;   /* UR target */
            aptl[2].x =   0; aptl[2].y =   0;   /* LL source */
            aptl[3].x =  19; aptl[3].y =  30;   /* UR source */
            GpiBitBlt(hps, hpsMemory, 4L, aptl, ROP_SRCCOPY, 0L);

                                               /* draw LR corner suit */
            aptl[0].x = 256; aptl[0].y = 104;   /* UL target */
            aptl[1].x = 275; aptl[1].y =  74;   /* LR target */
            aptl[2].x =   0; aptl[2].y =   0;   /* LL source */
            aptl[3].x =  19; aptl[3].y =  30;   /* UR source */
            GpiBitBlt(hps, hpsMemory, 4L, aptl, ROP_SRCCOPY, 0L);

                                               /* load number bitmap */
            hbmNumber = GpiLoadBitmap(hpsMemory, NULL, ID_ABMP, 0L, 0L);
            GpiSetBitmap(hpsMemory, hbmNumber);   /* set number bitmap */

                                               /* draw UL corner number */
            aptl[0].x = 120; aptl[0].y = 255;   /* LL target */
            aptl[1].x = 139; aptl[1].y = 285;   /* UR target */
            aptl[2].x =   0; aptl[2].y =   0;   /* LL source */
            GpiBitBlt(hps, hpsMemory, 3L, aptl, ROP_SRCCOPY, 0L);

                                               /* draw LR corner number */
            aptl[0].x = 256; aptl[0].y =  60;   /* UL target */
            aptl[1].x = 275; aptl[1].y =  30;   /* LR target */
            aptl[2].x =   0; aptl[2].y =   0;   /* LL source */
            aptl[3].x =  19; aptl[3].y =  30;   /* UR source */
            GpiBitBlt(hps, hpsMemory, 4L, aptl, ROP_SRCCOPY, 0L);
            }
         WinEndPaint(hps);
         break;

         case WM_DESTROY:
            GpiDestroyPS(hpsMemory);           /* destroy memory PS */
            DevCloseDC(hdcMemory);             /* close memory DC */
            GpiDeleteBitmap(hbmNumber);        /* delete bitmaps */
            GpiDeleteBitmap(hbmSuit);
            break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                        /* NULL / FALSE */
   }
```

```
/* --------- */
/* CARDBLT.H */
/* --------- */

USHORT cdecl main(void);
```

```
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1

#define ID_HEARTBMP 20
#define ID_DIAMONDBMP 30
#define ID_SPADEBMP 40
#define ID_CLUBBMP 50

#define ID_ABMP 100
```

```
# -----------------
# CARDBLT Make file
# -----------------

cardblt.obj: cardblt.c cardblt.h
   cl -c -G2s -W3 -Zp cardblt.c

cardblt.res: cardblt.rc cardblt.h
   rc -r cardblt.rc

cardblt.exe: cardblt.obj cardblt.def
   link /NOD cardblt,,NUL,os2 slibce,cardblt
   rc cardblt.res cardblt.exe

cardblt.exe: cardblt.res
   rc cardblt.res
```

```
; -----------
; CARDBLT.DEF
; -----------

NAME      CARDBLT    WINDOWAPI

DESCRIPTION 'Blt a card'

PROTMODE
STACKSIZE   4096
```

```
/* ---------- */
/* CARDBLT.RC */
/* ---------- */

#include <os2.h>
#include "cardblt.h"

BITMAP ID_HEARTBMP heart.bmp
BITMAP ID_DIAMONDBMP diamond.bmp
BITMAP ID_SPADEBMP spade.bmp
BITMAP ID_CLUBBMP club.bmp
BITMAP ID_ABMP a.bmp

MENU ID_FRAMERC
   BEGIN
      SUBMENU "Suits", 100
         BEGIN
            MENUITEM "", 101, MIS_SEPARATOR
```

```
            MENUITEM "#20", ID_HEARTBMP, MIS_BITMAP
            MENUITEM "", 102, MIS_SEPARATOR
            MENUITEM "#30", ID_DIAMONDBMP, MIS_BITMAP
            MENUITEM "", 103, MIS_SEPARATOR
            MENUITEM "#40", ID_SPADEBMP, MIS_BITMAP
            MENUITEM "", 104, MIS_SEPARATOR
            MENUITEM "#50", ID_CLUBBMP, MIS_BITMAP
            MENUITEM "", 105, MIS_SEPARATOR
        END
    END
```

The memory device context and presentation space associated with it are created during processing of the WM_CREATE message. During WM_PAINT processing, the suit bitmap is loaded into this PS and set with GpiLoadBitmap and GpiSetBitmap. Then it's bit-blitted to three places on the card: the upper left corner; the center, where it's enlarged to twice its original size; and the lower right corner, where it's flipped upside down.

We treat the number bitmap similarly, except it doesn't need to be drawn in the center of the card.

You might think you could draw the bitmap in the upper left corner of the card, as is done in CARD, and then bit-blit it from the screen to the other locations. But what happens if the user has resized the window so the upper left corner of the card is outside the window? You will find yourself bit-blitting pieces of scroll bar or something else. This demonstrates how useful the memory device context can be; it isn't affected by the uncertainties of the screen display.

Images

Images are another form of bitmap. They are loaded from an array using the GpiImage function. Although images are considered one of the graphics primitives, they are very limited compared to bitmaps. Since they're stored in an array, they can't be created with the icon editor. They are monochrome only: one bit per pixel. And they can't be expanded or compressed when they're loaded. For these reasons images are seldom used. Oddly, the pixels in an image are numbered starting at the top left, in contrast to bitmaps, where pixel numbering starts at the bottom left.

CLIPPING

Clipping means cutting off part of a graphics picture that lies outside a certain area. Sometimes when you draw a picture, you want part of it to be hidden. Calculating which parts of your picture should be hidden would be

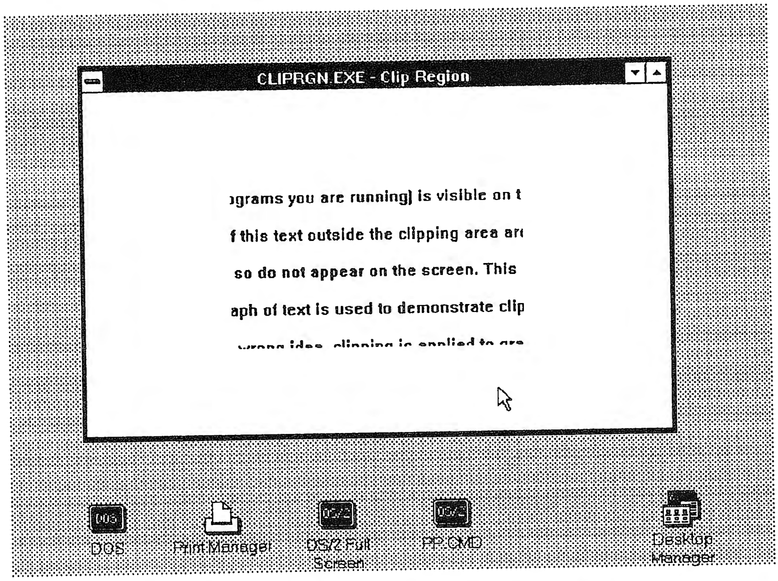


Figure 12-18. Output of the CLIPRGN program

tedious, but you can get PM to do it for you. Instead of drawing your picture to fit within certain boundaries, you draw it full size and let PM clip it to the desired size and shape.

In this section we'll show two methods of clipping: with regions and with paths. Since regions are created from rectangles, clipping regions are restricted to shapes that can be created from rectangles. Paths provide clipping to any shape you can draw, but operate more slowly.

Both example programs in this section demonstrate clipping using text, which is a convenient way to fill the screen with something recognizable, so we can see what's clipped and what isn't. However, clipping can be used in the same way on any graphics picture.

Clipping to Regions

This example program fills the window with text and then creates a clipping region that is half the width and half the height of the window. Only the text within the clipping region is visible, as shown in Figure 12-18.

Here are the listings for CLIPRGN.C, CLIPRGN.H, CLIPRGN, and CLIPRGN.DEF.

```
/* ---------------------------------- */
/* CLIPRGN.C - Using a clipping region */
/* ---------------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include <string.h>

#include "cliprgn.h"

HAB hab;                               /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                            /* handle for message queue */
   QMSG qmsg;                          /* message queue element */
   HWND hwnd, hwndClient;              /* handles for windows */

                                       /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);          /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);    /* create message queue */

                                       /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                       /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Clip Region", 0L, NULL, 0L, &hwndClient);

                                       /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);             /* destroy frame window */
   WinDestroyMsgQueue(hmq);            /* destroy message queue */
   WinTerminate(hab);                  /* terminate PM usage */

   return 0;
   }

                                       /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   static HPS hps;
   static RECTL rcl;
   static HRGN hrgn;
   static CHAR *pchString[NLINES] =
      {"This is a collection of character strings arranged as several",
       "lines of text. You, however, cannot see all the text, since it",
       "is clipped. Only the part of the text that is within the",
       "clipping area (actually region or path depending on which",
```

```
        "of the example programs you are running) is visible on the",
        "screen. All parts of this text outside the clipping area are",
        "clipped away, and so do not appear on the screen. This long",
        "and boring paragraph of text is used to demonstrate clipping.",
        "But, do not get the wrong idea, clipping is applied to graphics",
        "as well as text. After all, as we have seen in the previous",
        "chapter, text under PM is just another graphics primitive."};

   static POINTL ptlStart = {5 , 5};
   int i;
   static SIZEL sizl = {0, 0};
   static SIZEF sizefxBox;
   static HDC hdc;

   switch (msg)
      {
      case WM_PAINT:
         WinBeginPaint(hwnd, hps, NULL);
         GpiErase(hps);
         ptlStart.y = 5;
         for (i = NLINES; i-- > 0;      /* write text */
            ptlStart.y += FIXEDINT(sizefxBox.cy) * 2)
            {
            GpiCharStringAt(hps, &ptlStart, (LONG)strlen(pchString[i]),
               pchString[i]);
            }
         WinEndPaint(hps);
         break;

      case WM_SIZE:
         rcl.xLeft = SHORT1FROMMP(mp2) * .25;
         rcl.xRight = SHORT1FROMMP(mp2) * .75;
         rcl.yBottom = SHORT2FROMMP(mp2) * .25;
         rcl.yTop = SHORT2FROMMP(mp2) * .75;

         hrgn = GpiCreateRegion(hps, 1L, &rcl);   /* create region */
         GpiSetClipRegion(hps, hrgn, NULL);       /* set clipping region */
         break;

      case WM_CREATE:
         hdc = WinOpenWindowDC(hwnd);
         hps = GpiCreatePS(hab, hdc, &sizl, PU_PELS | GPIT_MICRO |
                  GPIA_ASSOC);
         GpiQueryCharBox(hps, &sizefxBox);
         break;

      case WM_DESTROY:
                                      /* no destroy for clipping region */
         GpiDestroyPS(hps);
                                      /* no close for window DC */
         WinDefWindowProc(hwnd, msg, mp1, mp2);
         break;

      case WM_ERASEBACKGROUND:
         return TRUE;
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }
```

```
   return NULL;                          /* NULL / FALSE */
   }
```

```
/* --------- */
/* CLIPRGN.H */
/* --------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define NLINES 11
```

```
# -----------------
# CLIPRGN Make file
# -----------------

cliprgn.obj: cliprgn.c cliprgn.h
   cl -c -G2s -W3 -Zp cliprgn.c

cliprgn.exe: cliprgn.obj cliprgn.def
   link /NOD cliprgn,,NUL,os2 slibce,cliprgn
```

```
; -----------
; CLIPRGN.DEF
; -----------

NAME       CLIPRGN   WINDOWAPI

DESCRIPTION 'Clipping to a region'

PROTMODE
STACKSIZE   4096
```

The text is drawn when the WM_PAINT message is received, using a loop to write successive strings from the array *pchString*.

The clipping region is created during processing of WM_SIZE. First a rectangle, *rcl*, is defined to be half the height and width of the window. Then a region is defined, using GpiCreateRegion, that is the same size as this rectangle. Finally, the clip region is set using GpiSetClipRegion.

Set the Clip Region

```
LONG GpiSetClipRegion(hps, hrgn, phrgn)
HPS hps             Handle to presentation space
HRGN hrgn           Handle of region to be made clipping region
PHRGN phrgn         Handle of previous clipping region

Returns: RGN_NULL, RGN_RECT, or RGN_COMPLEX if successful,
         RGN_ERROR if error
```

The second parameter specifies the new clipping region. The function fills in the third parameter with the handle of the old clipping region. Only one clipping region can exist at a given time. The coordinates for GpiSetClipRegion are device coordinates (pixels on the screen), not world coordinates (such as low metric).

A clipping region can be composed of several rectangles, just as other regions can be. The region to be made the clipping region can either be created with multiple rectangles, or several regions may be combined with GpiCombineRegion. The example uses a single rectangle.

In WM_CREATE we open a micro PS, which requires opening a window device context. Why not use a cached micro PS? We want to keep the presentation space for the duration of the program, so we can work with it in WM_SIZE as well as in WM_PAINT. In the current version of OS/2 WinEndPaint releases a cached micro PS even if it wasn't obtained with WinBeginPaint, so to keep the PS between messages, we create it with GpiCreatePS.

As we mentioned earlier, the second argument to WinBeginPaint is usually set to NULL, and the function returns a handle to a cached micro PS. Drawing to this PS only updated the invalid rectangle. In CLIPRGN, however, the presentation space is already allocated when we receive WM_PAINT, so WinBeginPaint doesn't need to obtain a PS. Its second argument is set to the handle of the micro PS obtained with GpiCreatePS. Thus, WinBeginPaint will set the update region of the PS to the invalid rectangle. Later, when we issue WinEndPaint, the update region of the PS is reset to its previous value.

When the program is about to terminate, and we receive a WM_DESTROY message, we don't destroy the region. Once a region has been made into the clipping region, it should not be changed in any way, including destroying it. Also, we don't close the device context, since a window device context should never be closed.

Clipping to Paths

Paths permit the area used for clipping to be any shape, not just a combination of rectangles. Our example program clips to a circle, thus providing a peephole effect, as shown in Figure 12-19.

In this example moving the mouse moves the peephole around on the window, so all the text can be viewed. (A similar effect could be used to provide a magnifying glass, using transformations, which we'll discuss in Chapter 13.)

Here are the listings for CLIPPATH.C, CLIPPATH.H, CLIPPATH, and CLIPPATH.DEF.

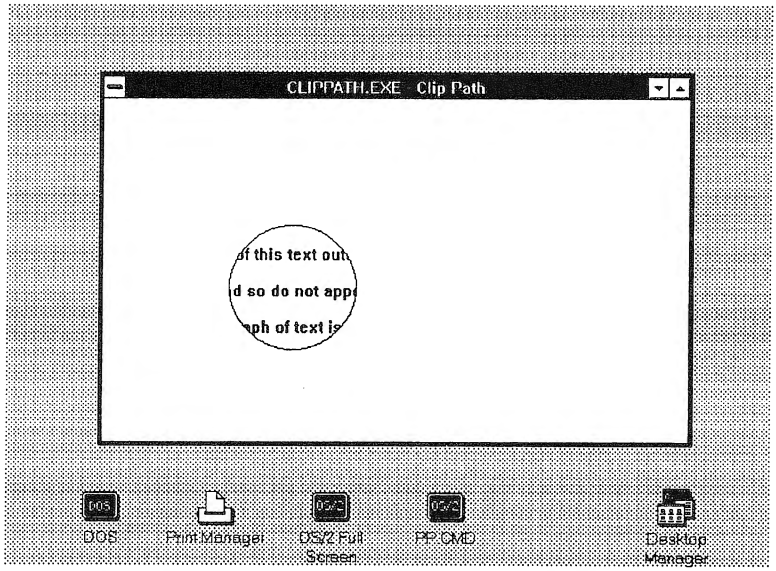


Figure 12-19. Output of the CLIPPATH program

```
/* ------------------------------- */
/* CLIPPATH.C - Clipping to a path */
/* ------------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include <string.h>

#include "clippath.h"

HAB hab;                            /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                            /* handle for message queue */
   QMSG qmsg;                          /* message queue element */
   HWND hwnd, hwndClient;              /* handles for windows */

                                       /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);          /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);    /* create message queue */

                                       /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW |
```

```
      CS_SYNCPAINT, 0);

                                      /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
            szClientClass, " - Clip Path", 0L, NULL, 0L, &hwndClient);

                                      /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);            /* destroy frame window */
   WinDestroyMsgQueue(hmq);           /* destroy message queue */
   WinTerminate(hab);                 /* terminate PM usage */

   return 0;
   }

                                      /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   static HPS hps;
   static CHAR *pchString[NLINES] =
      {"This is a collection of character strings arranged as several",
       "lines of text. You, however, cannot see all the text, since it",
       "is clipped. Only the part of the text that is within the",
       "clipping area (actually region or path depending on which",
       "of the example programs you are running) is visible on the",
       "screen. All parts of this text outside the clipping area are",
       "clipped away, and so do not appear on the screen. This long",
       "and boring paragraph of text is used to demonstrate clipping.",
       "But, do not get the wrong idea, clipping is applied to graphics",
       "as well as text. After all, as we have seen in the previous",
       "chapter, text under PM is just another graphics primitive."};
   static LONG lcbString[NLINES];
   static POINTL ptlStart = {5 , 5};
   int i;
   static SIZEL sizl = {0, 0};
   static SIZEF sizefxBox;
   static HDC hdc;
   static POINTL ptlCenter = {0,0};   /* center of magnifying glass */
   RECTL rcl;

   switch (msg)
      {
      case WM_PAINT:
         WinBeginPaint(hwnd, hps, &rcl);
         GpiConvert(hps, CVTC_DEVICE, CVTC_WORLD, 2L, &rcl);
         ptlStart.y = 5;
         for (i = NLINES; i-- > 0;      /* write text */
              ptlStart.y += FIXEDINT(sizefxBox.cy) * 2)
            {                           /* but, only in rcl */
            if (ptlStart.y + FIXEDINT(sizefxBox.cy) + 1 > rcl.yBottom &&
                ptlStart.y < rcl.yTop)
               GpiCharStringAt(hps, &ptlStart, lcbString[i], pchString[i]);
            }
         WinEndPaint(hps);
         break;

      case WM_MOUSEMOVE:
```

```
        WinSetPointer(HWND_DESKTOP, NULL);    /* remove pointer */

        ptlCenter.x = SHORT1FROMMP(mp1);
        ptlCenter.y = SHORT2FROMMP(mp1);
        GpiConvert(hps, CVTC_DEVICE, CVTC_WORLD, 1L, &ptlCenter);

        GpiErase(hps);

        GpiMove(hps, &ptlCenter);
        GpiBeginPath(hps, 1L);                /* define clip path */
        GpiFullArc(hps, DRO_OUTLINE, MAKEFIXED(200,0));
        GpiEndPath(hps);

        GpiSetClipPath(hps, 0L, SCP_RESET);
        GpiSetClipPath(hps, 1L, SCP_ALTERNATE | SCP_AND);

        GpiSetDrawControl(hps, DCTL_BOUNDARY, DCTL_ON);
        GpiResetBoundaryData(hps);
        GpiFullArc(hps, DRO_OUTLINE, MAKEFIXED(200,0));
        GpiSetDrawControl(hps, DCTL_BOUNDARY, DCTL_OFF);
        GpiQueryBoundaryData(hps, &rcl);
        GpiConvert(hps, CVTC_WORLD, CVTC_DEVICE, 2L, &rcl);
        WinInvalidateRect(hwnd, &rcl, FALSE);
        break;

    case WM_CREATE:
        hdc = WinOpenWindowDC(hwnd);
        hps = GpiCreatePS(hab, hdc, &sizl, PU_LOMETRIC | GPIT_MICRO |
                GPIA_ASSOC);
        GpiQueryCharBox(hps, &sizefxBox);
        for (i = 0; i < NLINES; i++)
            lcbString[i] = (LONG)strlen(pchString[i]);
        break;

    case WM_DESTROY:
                                    /* no destroy for clipping region */
        GpiDestroyPS(hps);
                                    /* no close for window DC */
        WinDefWindowProc(hwnd, msg, mp1, mp2);
        break;

    case WM_ERASEBACKGROUND:
        return TRUE;
        break;

    default:
        return(WinDefWindowProc(hwnd, msg, mp1, mp2));
        break;
    }

  return NULL;                      /* NULL / FALSE */
  }
```

```
/* ---------- */
/* CLIPPATH.H */
/* ---------- */
```

```
USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define NLINES 11
```

```
# ------------------
# CLIPPATH Make file
# ------------------

clippath.obj: clippath.c clippath.h
   cl -c -G2s -W3 -Zp clippath.c

clippath.exe: clippath.obj clippath.def
   link /NOD clippath,,NUL,os2 slibce,clippath
```

```
; ------------
; CLIPPATH.DEF
; ------------

NAME       CLIPPATH WINDOWAPI

DESCRIPTION 'Clipping to a path'

PROTMODE
STACKSIZE   4096
```

There are some subtleties in the CLIPPATH program, so let's examine it in detail.

As in the last example, a micro PS and a window device context are opened in the WM_CREATE message. Also, the size of the character box is determined, and the length of each text line is stored in the *lcbString* array.

There's quite a bit of code under the WM_MOUSEMOVE message. Some of it sets up the clipping path, and some is provided to make the program run faster.

First, since the peephole will act as the mouse pointer, the regular pointer is removed with WinSetPointer. Next the pointer location is obtained from the WM_MOUSEMOVE message's *mp1* and *mp2* parameters, and converted from device to world coordinates with GpiConvert.

GpiErase is used to erase the image that was under the previous location of the peephole. Now, after setting the current position to the mouse pointer, we define a path bracket with GpiBeginPath and GpiEndPath. The only graphics command in the path bracket is GpiFullArc, which defines the path as a circle.

The Clip Path

Now we can set the *clip path* (the area to be clipped) to the path bracket just defined. First, however, we must unset the old clip path. It's impossible to

move a clip path, so the old one must be destroyed and a new one created each time it's moved. GpiSetClipPath performs both these tasks.

Set the Clip Path

```
BOOL GpiSetClipPath(hps, idPath, cmdOptions)
HPS hps              Handle to presentation space
LONG idPath          1=clip path, 0=no clip path
LONG cmdOptions      Filling and combining modes (SCP_AND, etc.)

Returns: GPI_OK if successful, GPI_ERROR if error
```

The *idPath* argument is the identifier of the new path, which is always 1 in this version of OS/2. If a path is being reset (see the next argument), *idPath* is set to 0.

The *cmdOptions* argument can have one of the following values:

Identifier	Filling and Combining Modes
SCP_ALTERNATE	Use alternate mode to compute interior of clip path
SCP_WINDING	Use winding mode to compute interior of clip path
SCP_RESET	Release current clip path (*idPath* must be 0)
SCP_AND	Add specified path to current path

To set a new path, *idPath* is set to 1, and the SCP_AND option is used. Note that this adds the path just defined to the current clip path. That's why, to start a new clip path, you must reset the old one first, by calling GpiSetClipPath with *idPath*=0 and SCP_RESET.

Improving Clip Path Performance

As far as clipping goes, we could stop here. No further statements are necessary in WM_MOUSEMOVE, and in WM_PAINT we need only print out all the text as we did in the CLIPRGN example. The clip path would work as it should: We could move the peephole around with the mouse, and only the text within the peephole would be visible.

In fact, this is how our first version of the program was constructed. Unfortunately, performance was sluggish.

Our first attempt at improving performance involved invalidating only that portion of the window that needed to be redrawn. We reasoned that PM would be smart enough not to draw the text lying outside this region. Let's see how we do this.

The Update Rectangle

A window procedure receives a WM_PAINT message because some part of the window has become invalid and needs to be redrawn. The smallest rectangle that fits around this area is called the *update rectangle.*

In the CLIPPATH example the only thing that needs to be redrawn is the text inside the new position of the peephole. (The old peephole position is erased with GpiErase.) We reasoned that if we made the new peephole position the update rectangle, then PM would only draw the text inside this region. To cause the update rectangle to be set, we execute WinInvalidateRect, using the coordinates of the rectangle we want to make invalid. How do we find out the coordinates of the rectangle to be updated? This involves a new concept.

Boundary Rectangles

A *boundary rectangle* is the smallest rectangle that can be drawn around a specified graphics object. We want a boundary rectangle for the current peephole circle, so that we can make this region invalid with WinInvalidateRect. Since a circle is such a simple graphics shape, we could have constructed the boundary rectangle by hand, using the center and radius of the circle. However, it's more instructive to let PM create the boundary rectangle.

To do this, we first execute the GpiSetDrawControl function. This turns on a data collection process. It can be set to accumulate different kinds of data; in this case we want to record data about the boundary rectangle. All graphics objects drawn following GpiSetDrawControl will be checked to see how far they extend left, right, up, and down. The first object drawn will expand the boundary rectangle to its own maximum dimensions. If additional objects are drawn that extend further in any direction, then the boundary rectangle is expanded to these coordinates. Thus, as the drawing process proceeds, the boundary rectangle expands as necessary to bound the entire picture.

Set Current Draw Controls

```
BOOL GpiSetDrawControl(hps, lControl, flDraw)
HPS hps           Handle to presentation space
LONG lControl     Draw control to change (DCTL_BOUNDARY, etc.)
LONG flDraw       DCTL_ON or DCTL_OFF

Returns: GPI_OK if successful, GPI_ERROR if error
```

There are various drawing controls; the *lControl* argument to GpiSetDrawControl specifies which of them will be turned on or off. Here are the possibilities:

Identifier	**Drawing Control to**
DCTL_BOUNDARY	Accumulate boundary data
DCTL_CORRELATE	Correlate hits with pick aperture
DCTL_DISPLAY	Cause drawing to occur on device
DCTL_ERASE	Erase segments before drawing
DCTL_DYNAMIC	Update dynamic segments

Some of these options apply to concepts, such as the pick aperture and segments, that we'll look at in Chapter 13. We use DCTL_BOUNDARY here, since we're accumulating boundary rectangle data.

The *flDraw* argument can be either DCTL_ON, which turns on the drawing control, or DCTL_OFF, which turns it off. We first turn on the drawing control to start the data accumulation process.

Next we execute GpiResetBoundaryData. When we execute GpiSetDrawControl to accumulate boundary data, this data is added to any boundary data already accumulated; the boundary rectangle may already have a finite size as a result of previous operations. To start over with a boundary rectangle of zero size, we must reset the boundary data.

Reset Boundary Data

```
BOOL GpiResetBoundaryData(hps)
HPS hps           Handle to presentation space

Returns: GPI_OK if successful, GPI_ERROR if error
```

Now we draw the graphics object. In this case there is only one Gpi function: GpiFullArc, which draws the circle. Executing this function has two effects: It physically draws the circle on the screen, and it expands the boundary rectangle. Once our object is drawn, we turn off the drawing control using DCTL_OFF in GpiSetDrawControl.

We retrieve the control data for the boundary rectangle with GpiQueryBoundaryData.

Retrieve Current Boundary Data

```
BOOL GpiQueryBoundaryData(hps, prcl)
HPS hps            Handle to presentation space
PRECTL prcl        Boundary data structure

Returns: GPI_OK if successful, GPI_ERROR if error
```

This function returns the boundary rectangle in *prcl*. We convert the coordinates of this rectangle from world to device coordinates, and then call WinInvalidateRect to invalidate this specific rectangle. This generates the WM_PAINT message.

To summarize our attempt to improve performance: We found the bounding rectangle for the peephole and used this in WinInvalidateRect. We expected that during WM_PAINT, PM would redraw text only within the invalidated or update rectangle. However, despite our efforts, PM still delays significantly every time the peephole moves. It isn't smart enough to avoid making calculations on the invisible text. (It is smart enough to know it needn't do calculations for line segments that will be clipped away, but apparently the calculations for text are too complicated.)

Drawing Selected Text Lines

Since PM didn't significantly improve performance for us, we can optimize performance ourselves by drawing only those text lines that lie in the update region.

Although we have not used this fact in previous programs, the WinBeginPaint function returns the update rectangle in a structure of type RECTL in its third argument (which was set to NULL in previous examples). Thus, during WM_PAINT processing, we can figure out where the update rectangle is and avoid drawing text lines that don't fall within this rectangle.

We draw the text in somewhat the same way as in CLIPRGN, using a *for* loop; but within the loop, in the *if* statement, the update rectangle is used to decide whether a particular line of text will be written with GpiCharStringAt. The height of a character string was obtained in WM_CREATE using GpiQueryCharBox, and stored in *sizefxBox*. The variable *ptlStart.y* is the bottom of the character boxes on a text line. For a line of text to be drawn, the bottom of the characters must be below the top of the update rectangle, and the top of the characters must be above the bottom of the update rectangle.

With this addition to the program, performance is noticeably improved. We also could have figured out which characters in each line fell horizontally within the update region, but the improvement would not have been worth the increase in complexity.

Other Clipping Functions

Other functions exist to help with the clipping process. We'll have more to say about clipping when we talk about transformations in Chapter 13.

13

RETAINED GRAPHICS AND TRANSFORMATIONS

This chapter focuses on several related concepts: retained graphics, coordinate spaces, and transformations. Retained graphics, instead of being drawn immediately to the screen, are stored—that is, *retained*—in the presentation space. From the PS they can then be displayed or "played back" at any time. Both retained graphics and normal graphics can be changed in various ways before being displayed: They can be moved, made larger or smaller, rotated, and so on. These changes are called *transformations.* Transformations take place between one *coordinate space* and another. We'll discuss these topics and the relationships between them. We'll also show how to output a graphics image to a printer.

Do you need to use the concepts discussed in this chapter? For simple graphics images, possibly not. You can draw a bar chart, for example, without worrying about retain mode or transformations. But if you want to construct complex pictures as conveniently as possible, or if you want to perform animation or other advanced graphics activities, then you'll need to understand the concepts described here.

SEGMENTS AND THE PICTURE CHAIN

In our first example we'll create a car wheel. This object has several parts: a black tire, a red hubcap, and wire spokes between the tire and the hubcap.

(We'll add the rest of the car in future examples.) The program displays the wheel on the screen, but it doesn't do so in the straightforward ways we've seen in previous examples. Instead the wheel is first drawn in *retain* mode, so instead of being displayed, it's stored in the program's presentation space. It's stored as a graphics segment.

A *graphics segment* (or simply *segment*, if the context is clear) is a series of drawing instructions that creates a graphics object. It's like a graphics subprogram. (There's no connection between a graphics segment and the memory segments used in 80x86 architecture.) The segment is the primary unit of storage for retained graphics objects in the presentation space. A stored segment can be displayed or "played back" in a variety of ways.

When a number of segments are stored in the PS, they can form a *picture chain*, as shown in Figure 13-1. The ordering of segments on the chain determines the order in which they will be played back.

In the example program we store the car wheel as a segment. This segment becomes part of the picture chain. To display the wheel on the screen, we draw the entire chain, using the GpiDrawChain function. This displays all the segments in the chain. In this case there's only one segment in the chain: the wheel. Figure 13-2 shows the output of the WHEEL program.

Here are the listings for WHEEL.C, WHEEL.H, WHEEL, and WHEEL. DEF.

```
/* --------------------- */
/* WHEEL.C  Draw a wheel */
/* --------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "wheel.h"

HAB hab;                                /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd, hwndClient;               /* handles for windows */

                                        /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);     /* create message queue */
```

```
                                          /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                          /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Wheel", 0L, NULL, 0, &hwndClient);

                                          /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);                /* destroy frame window */
   WinDestroyMsgQueue(hmq);               /* destroy message queue */
   WinTerminate(hab);                     /* terminate PM usage */

   return 0;
   }

                                          /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                   MPARAM mp2)
   {
   static HDC hdc;
   static HPS hps;
   static SIZEL sizl = {0, 0};

   static POINTL ptl;
   int i;

   switch (msg)
      {
      case WM_PAINT:
         hps = WinBeginPaint(hwnd, hps, NULL);
         GpiErase(hps);

         GpiDrawChain(hps);               /* draw segment chain */

         WinEndPaint(hps);
         break;

      case WM_CREATE:
         hdc = WinOpenWindowDC(hwnd);         /* get a window DC */
                                       /* create & associate a normal PS */
         hps = GpiCreatePS(hab, hdc, &sizl, PU_LOENGLISH |
                   GPIT_NORMAL | GPIA_ASSOC);

         GpiSetDrawingMode(hps, DM_RETAIN);   /* set drawing mode */

                                              /* create wheel segment */
         GpiOpenSegment(hps, ID_WHEEL_SEG);   /* open segment */
         ptl.x = 150; ptl.y = 100;
         GpiMove(hps, &ptl);
         GpiSetColor(hps, CLR_BLACK);         /* outer tire */
         GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(55,0));
         GpiSetColor(hps, CLR_BACKGROUND);    /* spoke background */
         GpiFullArc(hps, DRO_FILL, MAKEFIXED(30,0));
         GpiSetColor(hps, CLR_RED);
         for(i=0; i < 20; i++)                /* inner tire and spokes */
            {
```

```
         GpiMove(hps, &ptl);
         GpiPartialArc(hps, &ptl, MAKEFIXED(30,0),
            MAKEFIXED(i*18,0), MAKEFIXED(18,0));
         }
      GpiMove(hps, &ptl);                   /* hubcap */
      GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(15,0));
      GpiCloseSegment(hps);                 /* close segment */

      GpiSetDrawingMode(hps, DM_DRAW);      /* set drawing mode */
      break;

   case WM_DESTROY:
      GpiDeleteSegment(hps, ID_WHEEL_SEG);  /* destroy segment */
      GpiAssociate(hps, NULL);              /* disassociate PS */
      GpiDestroyPS(hps);                    /* destroy PS */
      break;

   default:
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }

return NULL;                          /* NULL / FALSE */
}
```

```
/* ------- */
/* WHEEL.H */
/* ------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_WHEEL_SEG 200L
```

```
# ---------------
# Wheel Make file
# ---------------

wheel.obj: wheel.c wheel.h
   cl -c -G2s -W3 -Zp wheel.c

wheel.exe: wheel.obj wheel.def
   link /NOD wheel,,NUL,os2 slibce,wheel
```

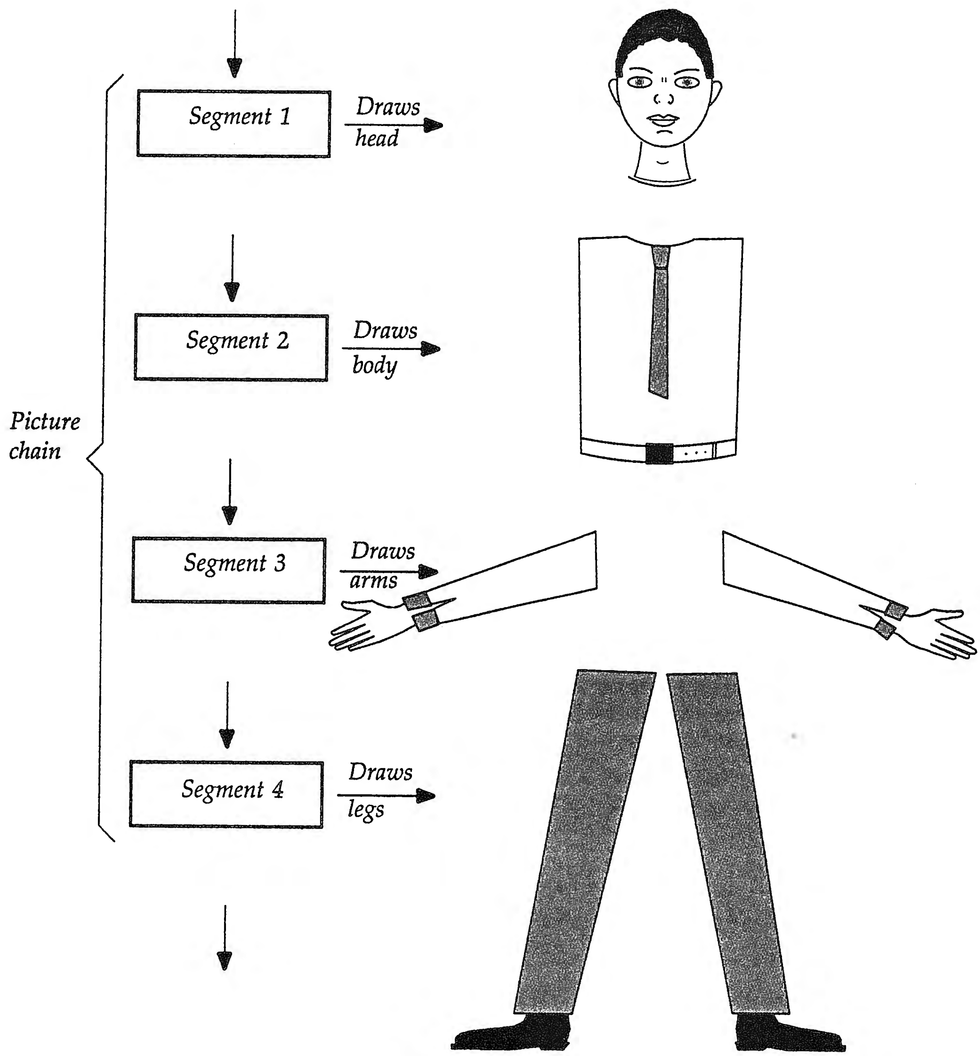


Figure 13-1. Segments and the picture chain

```
; ---------
; WHEEL.DEF
; ---------

NAME       WHEEL WINDOWAPI

DESCRIPTION 'Draw a wheel'

PROTMODE
STACKSIZE   4096
```

The Normal Presentation Space

Because we use retained graphics in this example, we must use a normal presentation space (as discussed in Chapter 10). We use the same APIs that we used to create the micro PS in previous examples. During WM_CREATE processing, a window device context is obtained with WinOpenWindowDC and associated with a PS created with GpiCreatePS. The major difference between obtaining a normal and a micro PS is that the GPIT_NORMAL identifier is used instead of GPIT_MICRO as the last argument of GpiCreatePS.

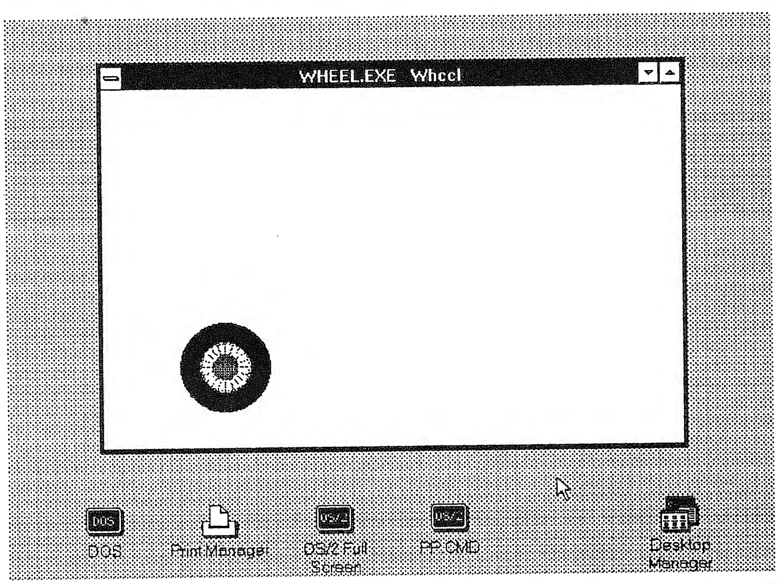


Figure 13-2. Output of the WHEEL program

Creating a Graphics Segment

Here are the steps necessary to create a graphics segment.

- Change to retain mode with GpiSetDrawingMode.
- Open the segment with GpiOpenSegment.
- Execute the Gpi calls to be stored in the segment.
- Close the segment with GpiCloseSegment.
- Restore draw mode (if necessary) with GpiSetDrawingMode.

In the WHEEL example these steps are carried out during WM_CREATE processing. Let's examine the APIs involved.

Setting the Drawing Mode

When you execute a series of Gpi functions to draw a graphics object, PM must know whether you intend these functions to actually display the picture on the screen (draw mode), store it in the presentation space (retain mode), or do both at the same time (draw-and-retain mode). Draw mode is the default; if you want anything else, you must use GpiSetDrawingMode to set the mode.

Set the Drawing Mode

```
BOOL GpiSetDrawingMode(hps, flMode)
HPS hps             Handle to presentation space
LONG flMode         Drawing mode

Returns: GPI_OK if successful, GPI_ERROR if error
```

The identifiers for the three drawing modes are

Identifier	Drawing Mode
DM_DRAW	Draw without retaining
DM_RETAIN	Retain without drawing
DM_DRAWANDRETAIN	Draw and retain

The drawing mode will remain in effect until reset.

Opening a Segment

A segment is opened with the GpiOpenSegment function.

Open a Segment

```
BOOL GpiOpenSegment(hps, idSegment)
HPS hps          Handle to presentation space
LONG idSegment   Segment identifier

Returns: GPI_OK if successful, GPI_ERROR if error
```

The programmer can give each segment a unique ID number. Our ID is ID_WHEEL_SEG. If a value of 0L is used for the segment ID, the segment will be an *unnamed segment*. An unnamed segment cannot be referenced separately, but it can be part of the picture chain. (More on this later.)

The GpiOpenSegment function resets all the primitive attributes (color, line type, and so on) to their default values. Thus, each segment starts off as a clean slate, as it were. If you want a primitive to have a certain attribute, you'll need to set it explicitly after you open the segment.

Following GpiOpenSegment, subsequent Gpi functions will be placed in the segment. Actually it's not the Gpi function itself that is placed in the segment, but a binary number that represents it. This binary number is called a *graphics order*.

For example, 25 02 04 00 is the graphics order representing a particular call to the GpiSetColor function. (All numbers are in hex.) The 25 is a code for GpiSetColor, 02 indicates that two more bytes of data follow, and 04 00 is the color value. Similarly, if a GpiLine function draws a line from the current position to the point (20, 30), the graphics order representing this function will be 81 08 20 00 00 00 30 00 00 00. The code for GpiLine is 81, and 08 indicates eight bytes of data follow. The next four bytes are the X position (which is type LONG: 32 bits), and the last four bytes represent the Y position.

A few Gpi functions will not work within a segment. These are functions that affect entire segments (such as GpiSetSegmentAttrs) or perform similar activities you probably wouldn't want to use within a segment anyway.

In WHEEL the Gpi functions in the segment create a car's wheel. GpiFullArc draws a black tire and a blank circle within the tire on which

the spokes will be placed, GpiPartialArc draws a series of 20 spokes, and finally, GpiFullArc draws a hubcap. The current position is moved as appropriate with GpiMove, and the colors are set with GpiSetColor. The codes for all these functions are stored in the PS as part of the segment.

Closing the Segment

When you're finished writing to the segment, you should close it with GpiCloseSegment.

Close Segment

```
BOOL GpiCloseSegment(hps)
HPS hps          Handle to presentation space

Returns: GPI_OK if successful, GPI_ERROR if error
```

When you know a segment will no longer be needed in a program, you should destroy it. In the example program this is done with GpiDeleteSegment when the WM_DESTROY message is received.

Delete Segment

```
BOOL GpiDeleteSegment(hps, idSegment)
HPS hps            Handle to presentation space
LONG idSegment     Segment identifier

Returns: GPI_OK if successful, GPI_ERROR if error
```

Editing Segments

We should mention that segments can be *edited.* Several APIs, like GpiDeleteElement, let you add and delete orders (which, as we noted, correspond to APIs) from a segment. This is useful when you want to make a small modification to an existing segment. You can also define a group of orders to be an *element.* Elements can be labeled, and then edited, as units.

The Picture Chain

Segments may be added to the picture chain when they are created. Segments have attributes, and one of the attributes is whether the segment will be chained or not. This attribute can be changed (as we'll see in the next example), but chained is the default. Thus, the segment created in WHEEL automatically becomes part of the chain. Segments that are part of the picture chain are called *chained* segments or *root* segments. Segments that are not part of the chain are called *unchained* segments or *callable* segments. Unchained segments are drawn by calling them from a root segment or directly from the program.

Segments are added to the chain in the order in which they are created. In WHEEL there is only one segment in the chain: ID_WHEEL_SEG.

The chain is displayed by executing GpiDrawChain. This is done during processing of the WM_PAINT message.

Draw Picture Chain

```
BOOL GpiDrawChain(hps)
HPS hps             Handle to presentation space

Returns: GPI_OK if successful, GPI_ERROR if error
```

When this function is executed, all the segments in the chain are played, that is, displayed. In the WHEEL example the wheel appears on the screen.

Functions related to GpiDrawChain are GpiDrawSegment, which draws a specific segment, and GpiDrawFrom, which draws the section of a picture chain between two specified segments. You can also change the order of the segments in the chain with GpiSetSegmentPriority.

Unchained Segments

A complete picture, as drawn on a graphics display or printer, may be made from several subpictures. For example, the segment created in the WHEEL example created a wheel, which is one part of a car. In the next example we're going to create the entire car. The car will be shown in side view, so two wheels are visible. One segment will hold the Gpi instructions

for the body of the car. We've already seen how one wheel (the rear one, as it turns out) is stored in a segment. The question is, how do we generate the other wheel?

We could create a second segment for the front wheel, using different coordinates. But this is wasteful: We would be storing the same set of graphics orders twice. It's far more effective to use the same segment to draw both wheels, but arrange to draw them in different locations on the screen. This is analogous to calling a subroutine or C function: It saves repeating unnecessary code and simplifies the program listing.

A segment that will be called several times must be an unchained segment. Using unchained segments effectively involves the ideas of coordinate spaces and transformations, but we'll delay a discussion of these topics until we've looked at the CAR example in detail.

Here are the listings for CAR.C, CAR.H, CAR, and CAR.DEF.

```
/* ----------------- */
/* CAR.C  Draw a car */
/* ----------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "car.h"

HAB hab;                                /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd, hwndClient;               /* handles for windows */

                                        /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);     /* create message queue */

                                        /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                        /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Car", 0L, NULL, 0, &hwndClient);

                                        /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);              /* destroy frame window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */
```

```
   return 0;
   }

                                          /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   static HDC hdc;
   static HPS hps;
   static SIZEL sizl = {0, 0};

   static POINTL ptl;
   int i;

   static POINTL ptlFenders[6] = {{145, 230}, {220, 160}, {220, 70},
                                  {450, 180}, {550, 210}, {545, 70}};

   static POINTL ptlWind[3] = {{265, 225}, {275, 225}, {310, 175}};

   static MATRIXLF matlfWheel2 = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                                  MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                                  330L, 0L, 1L};

   switch (msg)
      {
      case WM_PAINT:
         hps = WinBeginPaint(hwnd, hps, NULL);
         GpiErase(hps);

         GpiDrawChain(hps);              /* draw segment chain */

         WinEndPaint(hps);
         break;

      case WM_CREATE:
         hdc = WinOpenWindowDC(hwnd);
         hps = GpiCreatePS(hab, hdc, &sizl, PU_LOENGLISH |
                  GPIT_NORMAL | GPIA_ASSOC);

         GpiSetDrawingMode(hps, DM_RETAIN);  /* set drawing mode */

         GpiOpenSegment(hps, 0L);        /* open segment */
         GpiSetColor(hps, CLR_RED);      /* draw body rectangle */
         ptl.x = 125; ptl.y = 70;
         GpiMove(hps, &ptl);
         ptl.x = 505; ptl.y = 175;
         GpiBox(hps, DRO_OUTLINEFILL, &ptl, 0L, 0L);

         ptl.x = 300; ptl.y = 175;       /* draw windshield */
         GpiBeginArea(hps, BA_BOUNDARY);
         GpiMove(hps, &ptl);
         GpiPolyLine(hps, 3L, ptlWind);
         GpiEndArea(hps);

         ptl.x = 30; ptl.y = 70;         /* draw fenders */
         GpiMove(hps, &ptl);
         GpiSetColor(hps, CLR_DARKRED);
         GpiBeginArea(hps, BA_BOUNDARY);
         GpiPolySpline(hps, 6L, ptlFenders);
         GpiEndArea(hps);
                                         /* draw first wheel */
```

```
        GpiCallSegmentMatrix(hps, ID_WHEEL_SEG, 0L, NULL,
           TRANSFORM_REPLACE);

                                     /* draw second wheel */
        GpiCallSegmentMatrix(hps, ID_WHEEL_SEG, 7L, &matlfWheel2,
           TRANSFORM_REPLACE);

        GpiCloseSegment(hps);        /* close segment */

                                     /* create wheel segment */
        GpiOpenSegment(hps, ID_WHEEL_SEG);   /* open segment */
        ptl.x = 150; ptl.y = 100;
        GpiMove(hps, &ptl);
        GpiSetColor(hps, CLR_BLACK);         /* outer tire */
        GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(55,0));
        GpiSetColor(hps, CLR_BACKGROUND);    /* spoke background */
        GpiFullArc(hps, DRO_FILL, MAKEFIXED(30,0));
        GpiSetColor(hps, CLR_RED);
        for(i = 0; i < 20; i++)      /* inner tire and spokes */
           {
           GpiMove(hps, &ptl);
           GpiPartialArc(hps, &ptl, MAKEFIXED(30,0),
              MAKEFIXED(i*18,0), MAKEFIXED(18,0));
           }
        GpiMove(hps, &ptl);          /* hubcap */
        GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(15,0));
        GpiCloseSegment(hps);        /* close segment */

                                     /* unchain wheel segment */
        GpiSetSegmentAttrs(hps, ID_WHEEL_SEG, ATTR_CHAINED, ATTR_OFF);
        GpiSetDrawingMode(hps, DM_DRAW);   /* set drawing mode */
        break;

     case WM_DESTROY:
        GpiDeleteSegment(hps, ID_WHEEL_SEG);
        GpiAssociate(hps, NULL);
        GpiDestroyPS(hps);
        break;

     default:
        return(WinDefWindowProc(hwnd, msg, mp1, mp2));
        break;
     }

  return NULL;                       /* NULL / FALSE */
  }
```

```
/* ----- */
/* CAR.H */
/* ----- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_WHEEL_SEG 200L
```

```
# -------------
# CAR Make file
# -------------

car.obj: car.c car.h
   cl -c -G2s -W3 -Zp car.c

car.exe: car.obj car.def
   link /NOD car,,NUL,os2 slibce,car
```

```
; -------
; CAR.DEF
; -------

NAME        CAR WINDOWAPI

DESCRIPTION 'Draw a car'

PROTMODE
STACKSIZE   4096
```

Figure 13-3 shows the output of the program.

The architecture of this program is similar to that of the WHEEL example. A window device context and a normal presentation space are obtained

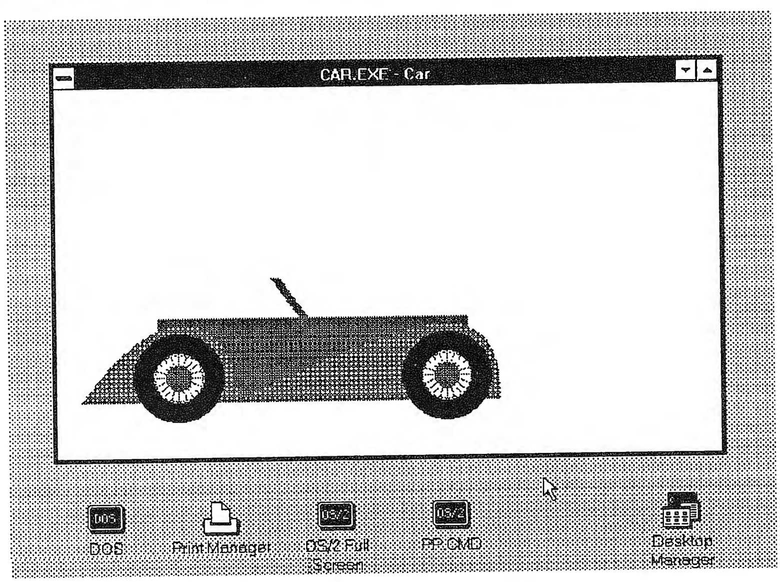


Figure 13-3. Output of the CAR program

during WM_CREATE processing. The picture chain is drawn during WM_PAINT processing with GpiDrawChain. The ID_WHEEL_SEG is also defined as it was in WHEEL, during WM_CREATE processing.

In CAR, however, a new segment is defined, one that creates the body of the car. This consists of a rectangle that forms the body, drawn with GpiBox; a polyline that forms the windshield, drawn with GpiPolyLine; and two splines that create the fenders, drawn with GpiPolySpline. The segment also contains APIs to set the color of these items. The windshield and fenders are drawn in area brackets to facilitate coloring them.

The GpiPolySpline Function

We discussed splines in Chapter 10, but didn't show an example. In CAR, splines are used to create the fenders of the car.

Draw Splines

```
LONG GpiPolySpline(hps, cptl, aptl)
HPS hps          Handle to presentation space
LONG cptl        Number of points in array
PPOINTL aptl     Array of structures for control points

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

Each spline requires two control points and two endpoints (see Figure 10-9). The first spline starts at the current position, and each additional spline starts at the endpoint of the one before. Thus, the array *aptl* must hold three points for each spline to be drawn. In CAR each of the two fenders is a spline, so there are six points in *aptl*, and *cptl* is set to 6L.

The GpiSetSegmentAttrs Function

After the wheel segment is defined, its chained attribute is turned off (set to unchained) with the GpiSetSegmentAttrs function. This function can switch several segment attributes on or off.

Set Segment Attributes

```
BOOL GpiSetSegmentAttrs(hps, idSegment, flAttribute, flAttrFlag)
HPS hps             Handle to presentation space
LONG idSegment      Segment identifier
LONG flAttribute    Attributes: ATTR_CHAINED, etc.
LONG flAttrFlag     On/off flag: ATTR_ON or ATTR_OFF

Returns: GPI_OK if successful, GPI_ERROR if error
```

The *idSegment* argument is set to the same segment ID that was supplied to the GpiOpenSegment API when the segment was created. The *flAttribute* argument can have one of the following values:

Identifier	**Attribute**
ATTR_CHAINED	Chained
ATTR_DETECTABLE	Detectability
ATTR_DYNAMIC	Dynamic
ATTR_FASTCHAIN	Fast chaining
ATTR_PROP_DETECTABLE	Propagate detectability
ATTR_PROP_VISIBLE	Propagate visibility
ATTR_VISIBLE	Visibility

Chained means that the segment is part of the picture chain. Detectable means the segment can be used in correlation operations, and dynamic refers to segments used for animation: They are drawn with the XOR mix mode to facilitate moving them. We'll see examples of correlation and animation in the next chapter. Fast chaining is used when all the primitive attributes are assumed to be the same from segment to segment. (In ordinary chaining, attributes are reset to their default values before each segment is drawn.) The detectable and visible attributes can be propagated—that is, set to apply to any segments called by the original segment (and segments called by called segments, and so on).

The *flAttrFlag* argument turns the selected attribute either on or off, using values of ATTR_ON or ATTR_OFF.

In CAR we turn off the chain attribute in the wheel segment. Turning off this attribute means that the segment is not part of the chain, and thus will not be automatically drawn when GpiDrawChain is executed. How then are the wheels drawn?

The function GpiCallSegmentMatrix, which is invoked twice after the car body segment has been defined, draws the wheels. The first time it draws the wheel without changing its original coordinates; this becomes the back wheel. The second time it changes the wheel's coordinates to move it to the right, thus drawing the front wheel. How does it change the coordinates?

To understand what's going on here, we need to examine the concepts of coordinate spaces and transformations.

COORDINATE SPACES AND TRANSFORMATIONS

A coordinate space is a place where you draw something. A transformation consists of moving, resizing, or similarly changing a graphics object. Transformations take place between one coordinate space and another. Let's look at coordinate spaces first, and then see how transformations are used to alter graphics objects between one coordinate space and the next.

Coordinate Spaces

When you draw a graphics object in PM, you draw it in world coordinate space. Think of a coordinate space as a piece of paper: It's a two-dimensional space in which you can draw graphics objects. If you execute GpiLine to draw a line 2 inches long, you can imagine the line being drawn in world coordinate space. That seems simple enough, but in PM there are actually four coordinate spaces: world space, model space, page space, and device space. Why so many spaces?

Suppose you're drawing a fairly complex picture. Say it's a street scene: There are cars in the foreground, pedestrians on the sidewalk, and trees and buildings in the background. Do you start right off drawing all these objects on the same coordinate space (piece of paper)? You could, but there are disadvantages.

First, the size of the objects in the final picture may not be the optimum size for drawing them. You may want to draw them quite large, and shrink them before placing them in the final picture. This is important if the user may at some point zoom (expand) a portion of the final picture to reveal additional detail. Second, and more important, you may want to rearrange parts of the picture after it's drawn. You may want a car to move down the street, for example. By drawing the car in a different coordinate space from the background, you can later add it to the final picture in different locations.

An analogous process is used for special effects in the movie business. Some objects (spaceships, for example) may be filmed against a black background and later combined with a background scene (perhaps stars and planets) that has been recorded on another piece of film.

The four different coordinate spaces in PM facilitate this multistep approach. The smallest parts of the picture are created in *world coordinate space.* These are items that will always remain as distinct units, such as the car's wheel in the CAR example, or a building's doors and windows. You may want to move the wheel as a whole, but you will never need to separate the tire and the spokes. You may move or resize a building's door, but you'll never change its knob, panels, or other details. These non-divisible items drawn in world coordinate space are called *subpictures.*

These subpictures, such as car wheels, car bodies, doors, and buildings, are combined into complete graphics objects or *models* in *model coordinate space.* A complete car or a complete building with doors and windows are typical objects in this space.

In *page coordinate space* (or more formally "presentation page coordinate space") the models from model space—cars, buildings, pedestrians, and trees—are combined into a complete picture.

Device coordinate space has a one-to-one correspondence with the hardware output device. Device coordinate space is device dependent, while the three other coordinate spaces are device independent. Often a picture is transformed unchanged from page space to device space (although it can be altered, as we'll see). Figure 13-4 shows the relationship of the four coordinate spaces.

Aside from device coordinate space, which is used for final display or printing, there are three levels for building up complex pictures from simple graphics objects: world, model, and page. This number three is somewhat arbitrary: One can imagine using four or more device-independent spaces, or only two. Three is a compromise between flexibility and excessive complexity.

As we'll discuss later, coordinate spaces are not quite real. However, it's important that you believe in them until you learn about transformations.

Transformations

Between each coordinate space and the next, you can specify a *transformation.* We apply a transformation to the car wheel when we move it to the right to draw the front wheel. This particular transformation takes place between world space and model space, but transformations also can be

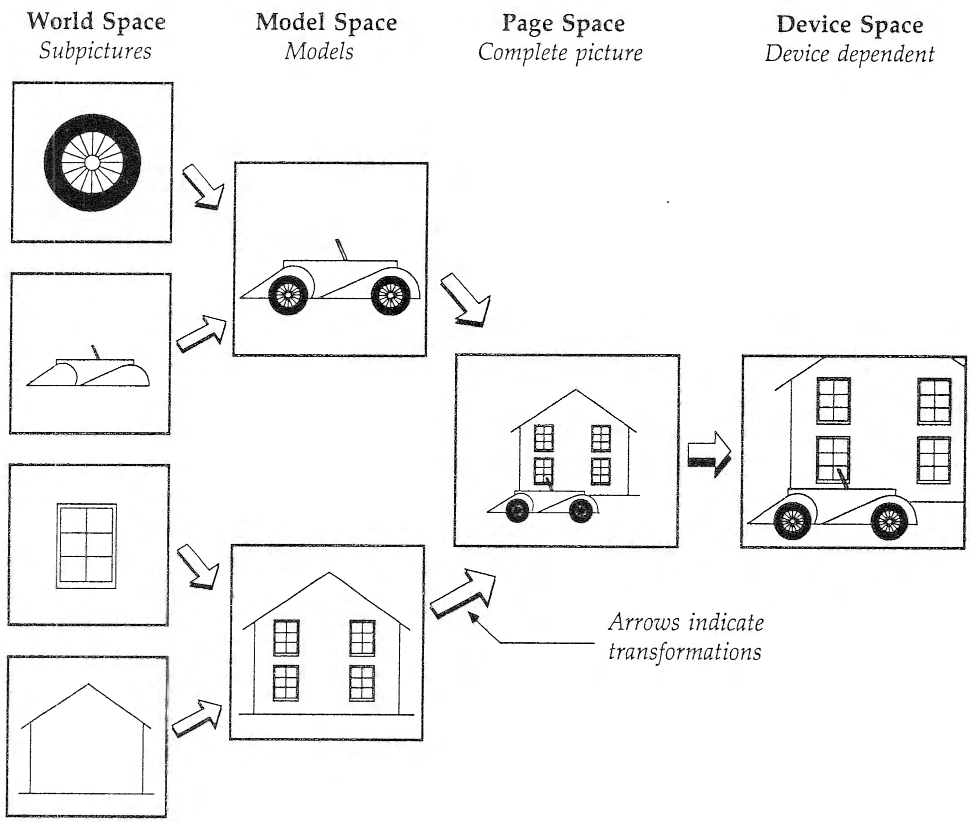


Figure 13-4. Coordinate spaces

applied between model and page space, and between page and device space. Transformations take place whether or not you specify them explicitly. The default transformation—which applies when no other transformation is specified—causes no change in the graphics object.

Transformations can cause one of six changes to a graphics object. The object can be scaled (expanded or contracted in the X or Y direction), rotated, translated (moved in the X or Y direction), sheared, and reflected.

Scaling can be used to zoom in or out on a graphics object, making it larger or smaller. *Rotation* is used to display objects at an angle, such as a car parked on a slope. *Translation* moves an object, such as a car driving across the screen. Different translations also can be used when drawing an object, so that it appears in different places; that's how we draw two wheels

on the car. *Shearing* changes the appearance of an object, as we saw with text characters in Chapter 11. *Reflection* makes mirror images, turning an object left-to-right or top-to-bottom.

Different Gpi functions are used to effect transformations, depending on whether the transformation is from world to model, model to page, or page to device space. Also, different functions are used for different purposes. Figure 13-5 summarizes the different transformations.

There are six kinds of transformations. From world to model space there are three: *model, segment,* and *instance.* From model to page space there are two: *viewing* and *default viewing;* and from page to device space there is only one: the *device* transformation.

Although in this chapter we're going to demonstrate transformations using graphics objects such as cars and trees, you should keep in mind that transformations apply equally well to characters. Provided it is generated by a vector font (not a bitmapped font), you can scale text, rotate it, shear it, and perform any of the other transformations on it.

The Transformation Matrix

You can use transformations without knowing anything about algebra, but a little background may make things clearer.

A transformation involves mapping every point in one space into a point in another space. If we're moving (translating) an object from world to model space, for example, any point (x, y) in world space will be moved by the same amount to arrive at the point (x', y') in model space. Expressed algebraically, this can be written

$$x' = x + Tx$$
$$y' = y + Ty$$

where Tx and Ty are the distances the object is to be moved in the X and Y directions. This is shown in Figure 13-6.

Transformation Equations

To be able to scale, rotate, and perform the other kinds of transformations, these equations must be made more general; x' must depend on y as well as x, and so on. They can be written in general form as

$$x' = Ax + Cy + E$$
$$y' = Bx + Dy + F$$

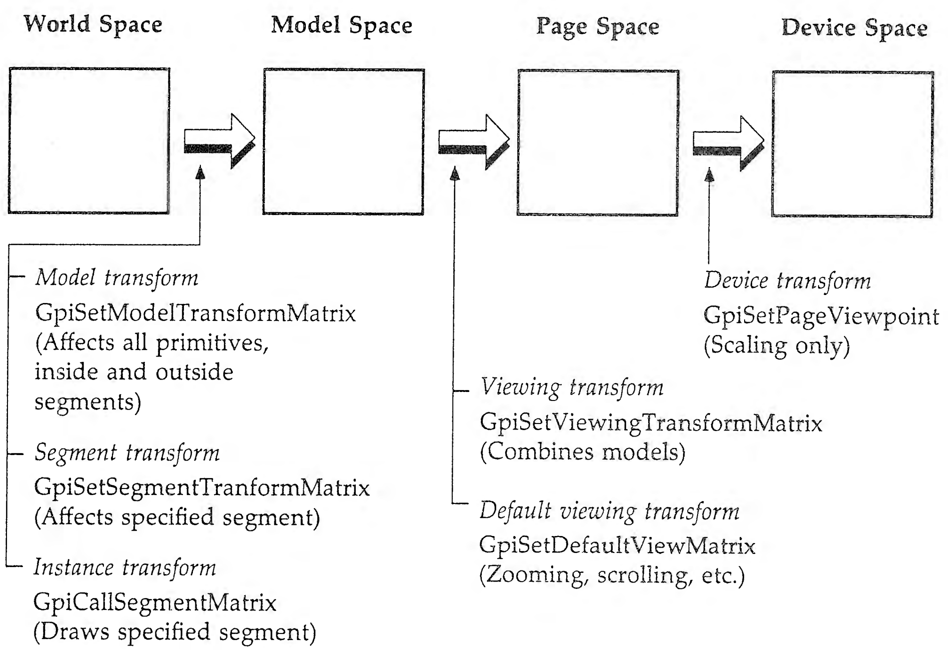


Figure 13-5. Transformations

By selecting the coefficients A, B, C, D, E, and F appropriately, the formulas can accomplish any of the transformations mentioned. We've already seen that translation is carried out when A and D are 1, C and B are 0, and E and F are the distances to move the object in the X and Y directions. To scale an object, A and D are set to appropriate scale factors, and B, C, E, and F are 0. To rotate an object through an angle θ, $A=\cos\theta$, $C=-\sin\theta$, $B=\sin\theta$, $D=\cos\theta$, and E and F are 0.

Matrix Notation

The APIs that perform transformations use the concept of a *matrix* to express the coefficients A, B, C, D, E, and F. A matrix is simply a compact way of writing these coefficients. If you don't know anything about matrix arithmetic, don't worry. All you need to know is how to plug values into the appropriate places in a structure that represents a matrix.

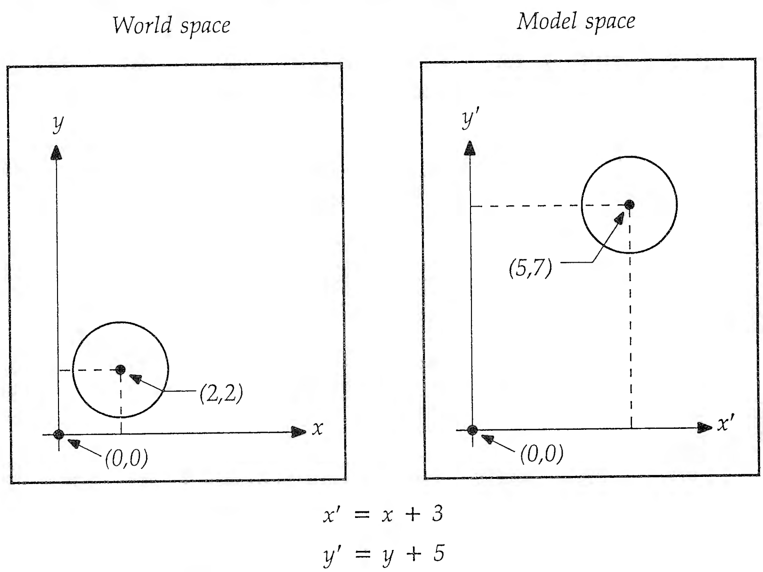


Figure 13-6. A translation transformation

Here's how the two general equations shown above are expressed as a matrix:

$$\begin{vmatrix} A & B & 0 \\ C & D & 0 \\ E & F & 1 \end{vmatrix}$$

Since this kind of matrix must be square, the six coefficients are increased to nine by using 0 and 1 in appropriate places. In PM notation this matrix is expressed as

$$\begin{vmatrix} M11 & M12 & M13 \\ M21 & M22 & M23 \\ M31 & M32 & M33 \end{vmatrix}$$

In the CAR program the second GpiCallSegmentMatrix function uses *matlfWheel2* as an argument. This argument is a variable of type MATRIXLF, which is defined this way in PMGPI.H:

```
typedef struct _MATRIXLF {
   FIXED fxM11;
   FIXED fxM12;
   LONG lM13;
   FIXED fxM21;
   FIXED fxM22;
   LONG lM23;
   LONG lM31;
   LONG lM32;
   LONG lM33;
} MATRIXLF;
```

This is another way to express the same coefficients as before, but using the appropriate data types and Hungarian notation.

The different kinds of transformation—scaling, rotation, and so on—use the nine coefficients in different ways. Figure 13-7 shows the matrices for the six transformations. It also includes a matrix for the identity (or unity) transformation, which leaves the object unaltered.

When you construct the MATRIXLF structure, it's useful to have the coefficients arranged in tabular form, so you can simply insert them into the appropriate lines of the structure.

Coefficient	Scaling	Rotation	Translation	Vert shear	Horiz shear	Reflection	Identity
fxM11	Sx	cosθ	1	1	Hx	Rx	1
fxM12	0	−sinθ	0	0	0	0	0
lM13	0	0	0	0	0	0	0
fxM21	0	sinθ	0	Vx	Hy	0	0
fxM22	Sy	cosθ	1	Vy	1	Ry	1
lM23	0	0	0	0	0	0	0
lM31	0	0	Tx	0	0	0	0
lM32	0	0	Ty	0	0	0	0
lM33	1	1	1	1	1	1	1

Setting up the MATRIXLF structure for a particular transformation is simply a matter of plugging in the values from the appropriate column. For example, to translate an object in the X direction, you would use the values from the third column. Set *lM31* to the distance you want to move the object, and *lM32* to 0 (since the object will not move in the Y direction). The other values in the MATRIXLF structure are set to the appropriate 0 and 1 values from the third column.

Note that the values for M11, M12, M21, and M22 are of type FIXED. You can use the MAKEFIXED macro to combine integer and fractional parts for the FIXED data types.

No matter what type of transformation is being used, the values for lM13 and lM23 are always 0, and lM33 is always 1.

$$\begin{vmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{vmatrix}$$
Scaling

$$\begin{vmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix}$$
Rotation

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{vmatrix}$$
Translation

$$\begin{vmatrix} 1 & 0 & 0 \\ Vx & Vy & 0 \\ 0 & 0 & 1 \end{vmatrix}$$
Vertical shear

$$\begin{vmatrix} Hx & 0 & 0 \\ Hy & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$
Horizontal shear

$$\begin{vmatrix} Rx & 0 & 0 \\ 0 & Ry & 0 \\ 0 & 0 & 1 \end{vmatrix}$$
Reflection

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$
Identity

Figure 13-7. The transformation matrices

Sx and Sy are horizontal and vertical scale factors. A value of 1 causes no scaling, 2 doubles the size, 0.5 makes it half the size, and so on. A 0.5 value would be expressed as MAKEFIXED(0, 0x8000). Hex is a convenient numbering system for the fractional part of a FIXED quantity, since its maximum value (plus 1) is an awkward 65,536 in decimal, but a nice round 0x10000 in hex.

Sine and cosine values are used for rotation, again expressed as FIXED values. In translation, Tx and Ty are the distances the object is to be moved in the X and Y directions, expressed as LONG values. We use an appropriate value of Tx in the CAR example to translate the wheel.

Vx and Vy are the values for vertical shear. Hx and Hy are the values for horizontal shear. Rx is set to reflect an object about the X axis, while Ry reflects it about the Y axis. These reflection values are always negative.

The Identity Matrix

If M11, M22, and M33 are 1, and all other matrix values are 0, the matrix is called the *identity matrix* or the *unity matrix*. A matrix with these values

performs no changes: The object looks just the same in the target space as it did in the source space. This is the default for all transformations unless explicitly changed by the program.

World-to-Model Transformations

In the CAR program we use one of the world-to-model transformations: the *instance* transformation. Let's examine that transformation and look at two other transformations that work from world to model space.

Instance Transformations

The transformation in CAR uses the GpiCallSegmentMatrix function to translate the car wheel to a different location. This function "calls" a segment as a subroutine, in the same way a program calls a C function. GpiCallSegmentMatrix performs a transformation and at the same time actually draws the segment. It's called an *instance* transformation.

Draw Segment Using Instance Transform

```
LONG GpiCallSegmentMatrix(hps, idSegment, cElements, pmatlf, lType)
HPS hps           Handle to presentation space
LONG idSegment    Segment identifier
LONG cElements    Number of valid matrix elements (0 to 9)
PMATRIXLF pmatlf  Matrix
LONG lType        Transformation modifier

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

The *idSegment* argument is the ID of the segment to be transformed and drawn, as specified in the GpiOpenSegment call that created it.

The *cElements* argument tells how many of the nine matrix elements are valid. This varies, depending on the type of transformation, since not all elements are needed. For example, if *cElements* is set to 5L, only the first five elements of the specified matrix will be used. The remaining four elements are taken from the identity matrix; they cause no change in the object. In the CAR example we use GpiCallSegmentMatrix twice. The first time we don't want to move the wheel, just draw it, so *cElements* is set to 0L. In this case all the matrix elements will come from the identity matrix. The second time we call GpiCallSegmentMatrix we want to move the wheel, so we

set the matrix up for a translation. The only matrix element to be altered is M31, which is set to the distance (330 hundredths of an inch) that we want to move the wheel in the X direction. The last two matrix elements are not used, so *cElements* can be set to 7L. (Since M33 is always 1, there's never any reason for *cElements* to be larger than 8L.)

The *pmatlf* argument points to the matrix itself, and *lType* can be one of the following:

Identifier	Transform Interaction
TRANSFORM_ADD	Add model transform to instance transform
TRANSFORM_PREEMPT	Add instance transform to model transform
TRANSFORM_REPLACE	Replace model transform with instance transform

We want to replace the existing transform with the one we've just defined, so we use TRANSFORM_REPLACE. (More on this in a moment.)

In instance transformations, the transformation is specific to the particular call to GpiCallSegmentMatrix. When the function returns, the transformation is no longer in effect. This is not the case with the other two kinds of world-to-model transformations.

Model Transformations

The second kind of world-to-model transformation is called a *model transformation* and is set by GpiSetModelTransformMatrix. This function has more far-reaching effects than did GpiCallSegmentMatrix. Once you specify this transformation, it's in effect for everything that's drawn from then on. It applies to all graphics primitives, whether they are inside a segment or not.

The various kinds of transform interaction, such as TRANSFORM_REPLACE (mentioned earlier for GpiCallSegmentMatrix), specify how the new model transform will be combined with the existing model transform. The new model transform can replace the existing one, or they can be combined, with either the new or the existing transformation coming first.

Note that the order in which transformations are combined makes a difference in the result. For example, translating an object and then rotating it may position the object differently than rotating it and then translating it.

Segment Transformations

The third kind of world-to-model transformation is the *segment* transformation, which is carried out with GpiSetSegmentTransformMatrix (the

transformation-oriented Gpi names do get rather long). A segment transformation, as its name implies, is performed on the graphics primitives in a specified segment. The segment transform would be used to transform only part of a graphics object (wheels on a car or ears on a face, for example) on its way from world to model space.

Model-to-Page Transformations

Now let's look at transformations from model to page space. Two APIs perform translations between these spaces, each for a different purpose. Our next example, SCENE, uses both these functions. This program creates the same car as in the previous example, but adds a tree. In addition, the program—using transformations—permits the user to zoom in and out, making the scene larger and smaller, and to scroll the scene left and right. The output when the program is first started is shown in Figure 13-8.

Here are the listings for SCENE.C, SCENE.H, SCENE, SCENE.DEF, and SCENE.RC.

```
/* -------------------------- */
/* SCENE.C  Draw car and tree */
/* -------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "scene.h"

HAB hab;                                /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd, hwndClient;               /* handles for windows */

                                        /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_MENU;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);     /* create message queue */

                                        /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                        /* create standard window */
```

```
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Scene", 0L, NULL, ID_FRAMERC, &hwndClient);

                                        /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);              /* destroy frame window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */

   return 0;
   }

                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   static HDC hdc;
   static HPS hps;
   static SIZEL sizl = {0, 0};

   static POINTL ptl;
   int i;

   static POINTL ptlFenders[6] = {{145, 230}, {220, 160}, {220, 70},
                                  {450, 180}, {550, 210}, {545, 70}};

   static POINTL ptlWind[3] = {{265, 225}, {275, 225}, {310, 175}};

   static MATRIXLF matlfWheel2 = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                                  MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                                  330L, 0L, 1L};

   static MATRIXLF matlfTree = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                                MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                                500L};

   static POINTL branch[BRANCHES] = {{150,330}, {150,300}, {120,270},
                                     {180,240}, {110,210}, {190,180},
                                     {80,120}, {220,130}, {150,140}};

   static MATRIXLF matlfZoomIn = {MAKEFIXED(2, 0), MAKEFIXED(0, 0), 0L,
                                  MAKEFIXED(0, 0), MAKEFIXED(2, 0)};

   static MATRIXLF matlfZoomOut =
                               {MAKEFIXED(0, 0x8000), MAKEFIXED(0, 0), 0L,
                                MAKEFIXED(0, 0), MAKEFIXED(0, 0x8000)};

   static MATRIXLF matlfLeft = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                                MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                               -100L};

   static MATRIXLF matlfRight = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                                 MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                                 100L};

   static MATRIXLF matlfInitView =
                          {MAKEFIXED(0, 0x2000), MAKEFIXED(0, 0), 0L,
                           MAKEFIXED(0, 0), MAKEFIXED(0, 0x2000)};
```

```
switch (msg)
   {
   case WM_PAINT:
      hps = WinBeginPaint(hwnd, hps, NULL);
      GpiErase(hps);

      GpiDrawChain(hps);            /* draw segment chain */

      WinEndPaint(hps);
      break;

   case WM_COMMAND:
      switch (COMMANDMSG(&msg) -> cmd)
         {
         case ID_ZOOMIN:
            GpiSetDefaultViewMatrix(hps, 5L, &matlfZoomIn,
               TRANSFORM_ADD);
            break;
         case ID_ZOOMOUT:
            GpiSetDefaultViewMatrix(hps, 5L, &matlfZoomOut,
               TRANSFORM_ADD);
            break;
         case ID_LEFT:
            GpiSetDefaultViewMatrix(hps, 7L, &matlfLeft, TRANSFORM_ADD);
            break;
         case ID_RIGHT:
            GpiSetDefaultViewMatrix(hps, 7L, &matlfRight,
               TRANSFORM_ADD);
            break;
         }
      WinInvalidateRect(hwnd, NULL, FALSE);
      break;

   case WM_CREATE:
      hdc = WinOpenWindowDC(hwnd);
      hps = GpiCreatePS(hab, hdc, &sizl, PU_LOENGLISH |
               GPIT_NORMAL | GPIA_ASSOC);

      GpiSetDrawingMode(hps, DM_RETAIN);  /* set drawing mode */

      GpiOpenSegment(hps, 0L);            /* open segment */
      GpiSetColor(hps, CLR_RED);          /* draw body rectangle */
      ptl.x = 125; ptl.y = 70;
      GpiMove(hps, &ptl);
      ptl.x = 505; ptl.y = 175;
      GpiBox(hps, DRO_OUTLINEFILL, &ptl, 0L, 0L);

      ptl.x = 300; ptl.y = 175;           /* draw windshield */
      GpiBeginArea(hps, BA_BOUNDARY);
      GpiMove(hps, &ptl);
      GpiPolyLine(hps, 3L, ptlWind);
      GpiEndArea(hps);

      ptl.x = 30; ptl.y = 70;       /* draw fenders */
      GpiMove(hps, &ptl);
      GpiSetColor(hps, CLR_DARKRED);
      GpiBeginArea(hps, BA_BOUNDARY);
      GpiPolySpline(hps, 6L, ptlFenders);
      GpiEndArea(hps);

      GpiCallSegmentMatrix(hps, ID_WHEEL_SEG, 0L, NULL,
```

```
        TRANSFORM_REPLACE);
    GpiCallSegmentMatrix(hps, ID_WHEEL_SEG, 7L, &matlfWheel2,
        TRANSFORM_REPLACE);

    GpiCloseSegment(hps);          /* close segment */

                                   /* create wheel segment */
    GpiOpenSegment(hps, ID_WHEEL_SEG);  /* open segment */
    ptl.x = 150; ptl.y = 100;
    GpiMove(hps, &ptl);
    GpiSetColor(hps, CLR_BLACK);        /* outer tire */
    GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(55,0));
    GpiSetColor(hps, CLR_BACKGROUND);   /* spoke background */
    GpiFullArc(hps, DRO_FILL, MAKEFIXED(30,0));
    GpiSetColor(hps, CLR_RED);
    for(i=0; i<20; i++)                 /* inner tire and spokes */
        {
        GpiMove(hps, &ptl);
        GpiPartialArc(hps, &ptl, MAKEFIXED(30,0),
            MAKEFIXED(i*18,0), MAKEFIXED(18,0));
        }
    GpiMove(hps, &ptl);                 /* hubcap */
    GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(15,0));
    GpiCloseSegment(hps);               /* close segment */

    GpiSetSegmentAttrs(hps, ID_WHEEL_SEG, ATTR_CHAINED, ATTR_OFF);

    GpiSetViewingTransformMatrix(hps, 7L, &matlfTree, TRANSFORM_REPLACE);

                                        /* create tree */
    GpiOpenSegment(hps, ID_TREE_SEG);   /* open segment */

    ptl.x = 135; ptl.y = 0;             /* draw trunk */
    GpiMove(hps, &ptl);
    ptl.x = 165; ptl.y = 100;
    GpiSetColor(hps, CLR_BROWN);
    GpiBox(hps, DRO_FILL, &ptl, 0L, 0L);

    for(i=0; i<BRANCHES; i++)           /* draw branches */
        {
        GpiMove(hps, &branch[i]);
        GpiSetColor(hps, CLR_DARKGREEN);
        GpiFullArc(hps, DRO_FILL, MAKEFIXED(50,0));
        GpiSetColor(hps, CLR_GREEN);
        branch[i].x +=10; branch[i].y += 10;
        GpiMove(hps, &branch[i]);
        GpiFullArc(hps, DRO_FILL, MAKEFIXED(30,0));
        }

    GpiCloseSegment(hps);               /* close segment */

    GpiSetDrawingMode(hps, DM_DRAW);    /* set drawing mode */

                                        /* set default view */
    GpiSetDefaultViewMatrix(hps, 5L, &matlfInitView, TRANSFORM_REPLACE);
    break;

case WM_DESTROY:
                                   /* delete segments */
    GpiDeleteSegments(hps, ID_WHEEL_SEG, ID_TREE_SEG);
    GpiAssociate(hps, NULL);
```

```
         GpiDestroyPS(hps);
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                            /* NULL / FALSE */
   }
```

```
/* ------- */
/* SCENE.H */
/* ------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_ZOOMIN 101
#define ID_ZOOMOUT 102
#define ID_LEFT 103
#define ID_RIGHT 104

#define ID_WHEEL_SEG 200L
#define ID_TREE_SEG 300L

#define BRANCHES 9
```

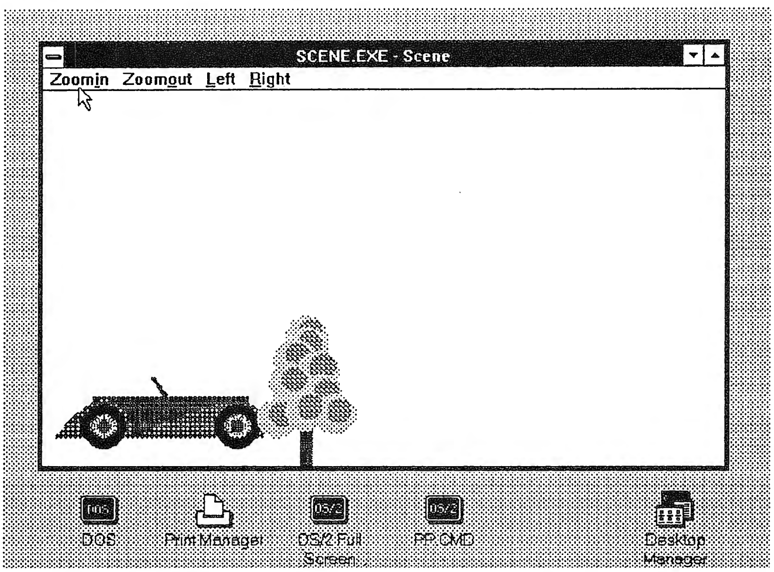


Figure 13-8. Output of the SCENE program

```
# ---------------
# SCENE Make file
# ---------------

scene.obj: scene.c scene.h
   cl -c -G2s -W3 -Zp scene.c

scene.res: scene.rc scene.h
   rc -r scene.rc

scene.exe: scene.obj scene.def
   link /NOD scene,,NUL,os2 slibce,scene
   rc scene.res

scene.exe: scene.res
   rc scene.res
```

```
; ---------
; SCENE.DEF
; ---------

NAME      SCENE WINDOWAPI

DESCRIPTION 'Draw a car and tree scene'

PROTMODE
STACKSIZE   4096
```

```
/* -------- */
/* SCENE.RC */
/* -------- */

#include <os2.h>
#include "scene.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "Zoom~in", ID_ZOOMIN
      MENUITEM "Zoom~out", ID_ZOOMOUT
      MENUITEM "~Left", ID_LEFT
      MENUITEM "~Right", ID_RIGHT
   END
```

Much of this program is derived from CAR. The segments that create the wheel and the car body are the same. The wheels are drawn using GpiCallSegmentMatrix as before. A new segment, with an ID of ID_TREE_SEG, creates the tree. The major change is the introduction of two new Gpi functions to perform transformations between model and page space. GpiSetDefaultViewMatrix zooms and scrolls the entire scene, and GpiSetViewingTransformMatrix moves the tree to an appropriate location relative to the car.

The GpiSetDefaultViewMatrix Function

The GpiSetDefaultViewMatrix function is used to zoom and scroll the entire scene. It's executed during WM_COMMAND processing, in response to the user making menu selections that send command values of ID_ZOOMIN, ID_LEFT, and so on.

Set Default Viewing Transformation

```
BOOL GpiSetDefaultViewMatrix(hps, cElements, pmatlf, flType)
HPS hps             Handle to presentation space
LONG cElements      Number of valid matrix elements (0 to 9)
PMATRIXLF pmatlf    Matrix
LONG flType         Transformation modifier

Returns: GPI_OK if successful, GPI_ERROR if error
```

The arguments to this function are similar to those to GpiCallSegmentMatrix. There is no *idSegment* argument, since this transformation applies to all graphics drawn after it's set, not just a particular segment.

The *flType* argument describes how the default viewing transformation specified in *pmatlf* is combined with the existing default viewing transformation.

Identifier	**Transform Interaction**
TRANSFORM_ADD	Add this to existing default viewing transform
TRANSFORM_PREEMPT	Add existing default viewing transform to this
TRANSFORM_REPLACE	Replace existing default viewing transform with this

We use TRANSFORM_ADD so that the specified matrix will be added to the existing one. If we've already zoomed in once on the car, and we zoom in again, we want the visual effect to be cumulative: If a zoom makes the scene twice as big, two zooms should make it four times as big. Zooms and scrolls can also be added to each other, so the scene may be simultaneously moved right, for example, and expanded.

For zooming, the M11 and M22 elements of the *matlfZoomIn* and *matlfZoomOut* matrices are set to 2.0 (zooming in) or 0.5 (zooming out). For

scrolling (which is in the X direction), the M31 elements of *matlfRight* and *matlfLeft* are set to 100L and –100L, respectively.

GpiSetDefaultViewMatrix is also used at the end of WM_CREATE processing to scale the initial size of the scene, using the *matlfInitView* matrix.

The GpiSetViewingTransformMatrix Function

As defined in ID_TREE_SEG, the tree is in the wrong place: in the middle of the car. We want it to appear near the front, so we need to move it. However, we don't want to move the car. What we want is a function that provides a transformation for a particular segment, without affecting other segments. The GpiSetViewingTransformMatrix does this.

Set Viewing Transformation

```
BOOL GpiSetViewingTransformMatrix(hps, cElements, pmatlf, flType)
HPS hps             Handle to presentation space
LONG cElements      Number of valid matrix elements (0 to 9)
PMATRIXLF pmatlf    Matrix
LONG flType         Transformation modifier

Returns: GPI_OK if successful, GPI_ERROR if error
```

With this function, only one value is valid for *flType:* TRANSFORM_REPLACE. The existing viewing transformation is always replaced by the one specified in *matlf.*

This function sets the viewing transformation for any segments opened after the function is executed. When a segment is opened, it is automatically given this transformation. Once given the transformation, it can't be changed; it becomes an unalterable attribute of the segment. This is inconvenient in some circumstances. For instance, if you want to transform the same segment from model space into multiple places in page space, you can't simply apply GpiSetViewingTransformMatrix over and over to the same segment.

The workaround for this is to put the object you want to replicate into a callable (unchained) segment. Suppose you have a car in model space, and you want to draw it into multiple places in page space. You put the car into a callable segment. Then, in the picture chain, each time you want to draw

the car in a different position, do the following: Open a segment, set the viewing transform with GpiSetViewingTransformMatrix, call the car segment with GpiCallSegmentMatrix, and close the segment.

Page-to-Device Transformations

The transformation from page to device, the *device transformation,* is an odd one. It doesn't use the usual MATRIXLF matrix, and the only kind of transformation it permits is scaling.

The device transformation scales the presentation page (a rectangle in page space) to a rectangle in device space called the *page viewport,* as shown in Figure 13-9.

The presentation page can be sized in two ways. You can size it explicitly when you create the presentation space with GpiCreatePS. To do this,

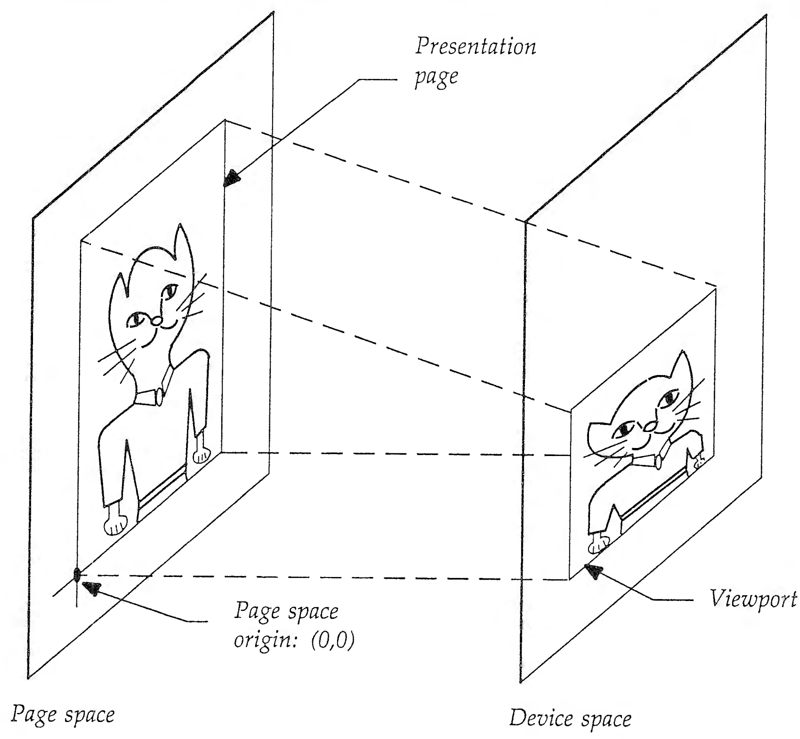


Figure 13-9. The device transformation

you set the *sizl* argument to appropriate coordinates. Or, if you use a size of (0, 0) for *sizl*, then PM will make up its own size for the presentation page. For this to happen, the presentation space must be associated with a device context. PM then uses the maximum size of the device—the paper size for printers and the maximized window size for the screen—as the presentation page size. In this example (as with others in this book), we let PM size the presentation page.

The page viewport is specified by the GpiSetPageViewport function.

Set Viewport in Device Space

```
BOOL GpiSetPageViewport(hps, prclViewport)
HPS hps                 Handle to presentation space
PRECTL prclViewport     Viewport rectangle (device units)

Returns: GPI_OK if successful, GPI_ERROR if error
```

The *prclViewport* argument to this function is a rectangle of type RECTL that specifies the coordinates of the page viewport. The images on the presentation page will be scaled to fit the page viewport. This can result in a change in aspect ratio as well as in size.

One common use for the device transform is scaling the presentation page to exactly fill a window. That's what our example program does. Here are the listings for VIEWPORT.C, VIEWPORT.H, VIEWPORT, and VIEWPORT.DEF.

```
/* ------------------------- */
/* VIEWPORT  Fill the window */
/* ------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "viewport.h"

HAB hab;                                /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd, hwndClient;               /* handles for windows */
```

```
                                    /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);       /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0); /* create message queue */

                                    /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                    /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
          szClientClass, " - Viewport", 0L, NULL, 0, &hwndClient);

                                    /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);          /* destroy frame window */
   WinDestroyMsgQueue(hmq);         /* destroy message queue */
   WinTerminate(hab);               /* terminate PM usage */

   return 0;
   }

                                    /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                  MPARAM mp2)
   {
   static HDC hdc;
   static HPS hps;
   static SIZEL sizl = {0, 0};

   static POINTL ptl;
   static RECTL rclViewport = {0, 0, 100, 100};
   int i;

   static POINTL ptlFenders[6] = {{145, 230}, {220, 160}, {220, 70},
                                  {450, 180}, {550, 210}, {545, 70}};

   static POINTL ptlWind[3] = {{265, 225}, {275, 225}, {310, 175}};

   static MATRIXLF matlfWheel2 = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                                  MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                                  330L, 0L, 1L};

   static MATRIXLF matlfTree = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                                MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                                500L};

   static POINTL branch[BRANCHES] = {{150,330}, {150,300},
                 {120,270}, {180,240}, {110,210}, {190,180},
                 {80,120}, {220,130}, {150,140}};

   switch (msg)
      {
      case WM_PAINT:
         hps = WinBeginPaint(hwnd, hps, NULL);
         GpiErase(hps);
```

```
      GpiDrawChain(hps);          /* draw segment chain */

      WinEndPaint(hps);
      break;

   case WM_CREATE:
      hdc = WinOpenWindowDC(hwnd);
      hps = GpiCreatePS(hab, hdc, &sizl, PU_LOENGLISH |
               GPIT_NORMAL | GPIA_ASSOC);

      GpiSetDrawingMode(hps, DM_RETAIN);  /* set drawing mode */

      GpiOpenSegment(hps, 0L);            /* open segment */
      GpiSetColor(hps, CLR_RED);          /* draw body rectangle */
      ptl.x = 125; ptl.y = 70;
      GpiMove(hps, &ptl);
      ptl.x = 505; ptl.y = 175;
      GpiBox(hps, DRO_OUTLINEFILL, &ptl, 0L, 0L);

      ptl.x = 300; ptl.y = 175;           /* draw windshield */
      GpiBeginArea(hps, BA_BOUNDARY);
      GpiMove(hps, &ptl);
      GpiPolyLine(hps, 3L, ptlWind);
      GpiEndArea(hps);

      ptl.x = 30; ptl.y = 70;             /* draw fenders */
      GpiMove(hps, &ptl);
      GpiSetColor(hps, CLR_DARKRED);
      GpiBeginArea(hps, BA_BOUNDARY);
      GpiPolySpline(hps, 6L, ptlFenders);
      GpiEndArea(hps);

      GpiCallSegmentMatrix(hps, ID_WHEEL_SEG, 0L, NULL,
         TRANSFORM_REPLACE);
      GpiCallSegmentMatrix(hps, ID_WHEEL_SEG, 7L, &matlfWheel2,
         TRANSFORM_REPLACE);

      GpiCloseSegment(hps);               /* close segment */

                                          /* create wheel segment */
      GpiOpenSegment(hps, ID_WHEEL_SEG);  /* open segment */
      ptl.x = 150; ptl.y = 100;
      GpiMove(hps, &ptl);
      GpiSetColor(hps, CLR_BLACK);        /* outer tire */
      GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(55,0));
      GpiSetColor(hps, CLR_BACKGROUND);   /* spoke background */
      GpiFullArc(hps, DRO_FILL, MAKEFIXED(30,0));
      GpiSetColor(hps, CLR_RED);
      for(i=0; i<20; i++)                 /* inner tire and spokes */
         {
         GpiMove(hps, &ptl);
         GpiPartialArc(hps, &ptl, MAKEFIXED(30,0),
            MAKEFIXED(i*18,0), MAKEFIXED(18,0));
         }
      GpiMove(hps, &ptl);                 /* hubcap */
      GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(15,0));
      GpiCloseSegment(hps);               /* close segment */

      GpiSetSegmentAttrs(hps, ID_WHEEL_SEG, ATTR_CHAINED, ATTR_OFF);

      GpiSetViewingTransformMatrix(hps, 7L, &matlfTree, TRANSFORM_REPLACE);
```

```
                                           /* create tree */
      GpiOpenSegment(hps, ID_TREE_SEG);    /* open segment */

      ptl.x = 135; ptl.y = 0;              /* draw trunk */
      GpiMove(hps, &ptl);
      ptl.x = 165; ptl.y = 100;
      GpiSetColor(hps, CLR_BROWN);
      GpiBox(hps, DRO_FILL, &ptl, 0L, 0L);

      for(i=0; i<BRANCHES; i++)      /* draw branches */
         {
         GpiMove(hps, &branch[i]);
         GpiSetColor(hps, CLR_DARKGREEN);
         GpiFullArc(hps, DRO_FILL, MAKEFIXED(50,0));
         GpiSetColor(hps, CLR_GREEN);
         branch[i].x +=10; branch[i].y += 10;
         GpiMove(hps, &branch[i]);
         GpiFullArc(hps, DRO_FILL, MAKEFIXED(30,0));
         }

      GpiCloseSegment(hps);                /* close segment */

      GpiSetDrawingMode(hps, DM_DRAW);     /* set drawing mode */

      break;

   case WM_SIZE:
      rclViewport.xRight = SHORT1FROMMP(mp2);
      rclViewport.yTop = SHORT2FROMMP(mp2);
                                 /* set viewport to entire window */
      GpiSetPageViewport(hps, &rclViewport);
      break;

   case WM_DESTROY:
      GpiDeleteSegments(hps, ID_WHEEL_SEG, ID_TREE_SEG);
      GpiAssociate(hps, NULL);
      GpiDestroyPS(hps);
      break;

   default:
      return(WinDefWindowProc(hwnd, msg, mp1, mp2));
      break;
   }

return NULL;                        /* NULL / FALSE */
}
```

```
/* ---------- */
/* VIEWPORT.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_WHEEL_SEG 200L
#define ID_TREE_SEG 300L

#define BRANCHES 9
```

```
# ------------------
# VIEWPORT Make file
# ------------------

viewport.obj: viewport.c viewport.h
   cl -c -G2s -W3 -Zp viewport.c

viewport.exe: viewport.obj viewport.def
   link /NOD viewport,,NUL,os2 slibce,viewport
```

```
; ------------
; VIEWPORT.DEF
; ------------

NAME   VIEWPORT WINDOWAPI

DESCRIPTION 'Fill the window using viewport'

PROTMODE
STACKSIZE   4096
```

The program starts with the scene containing the car and tree, as seen in previous examples. The device transformation takes place automatically. Whenever you resize the window, the presentation page is scaled in the X and Y directions to fit. Figure 13-10 shows what happens if you make the

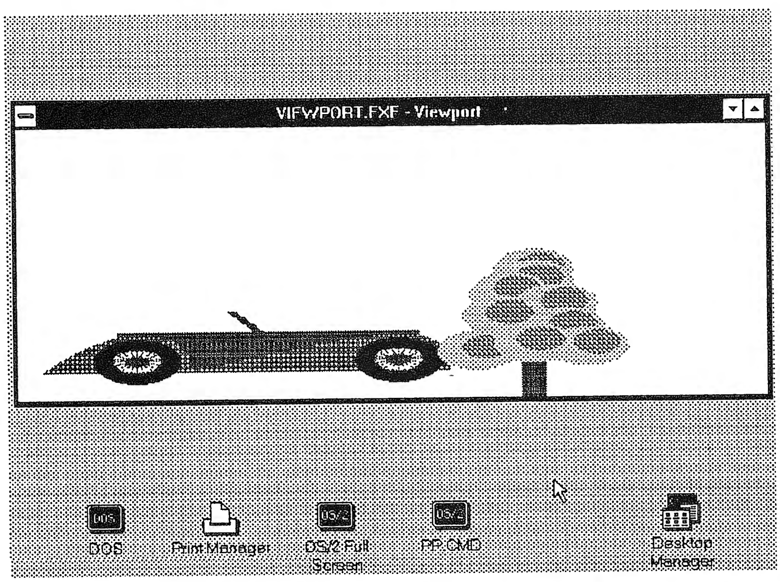


Figure 13-10. Output of the VIEWPORT program

window long but not very high: The car and tree also become long but not very high.

The viewport is set on receipt of a WM_SIZE message. The size of the window is derived from *mp2,* and GpiSetPageViewport sets the viewport to this size. Now no matter what size or shape you make the window, the program scales the image in the presentation page to fit.

That's all you need to do to size a scene to fit a window. Notice how much easier this is than manually resizing and redrawing all the components of a graphics image to fit the window every time its size is changed.

Coordinate Systems and Arbitrary Units

The coordinate systems for world, model, and page space are the same. They run from −134,217,728 to 134,217,727 in both the X and Y directions. These coordinates require 28 bits, so PM uses 32-bit integers of type LONG to hold them. The limits for device space, on the other hand, run from −32,768 to 32,767 (which require 16-bit integers).

We've already seen how the presentation space can be given different measurement units with GpiCreatePS: high English, low English, high metric, low metric, and twips. These are useful units when you want a printed image to be exactly the right size. For example, if you use GpiLine to specify a line 4 centimeters long, it will appear exactly this long on a printer or plotter. (It will be approximately this long on a typical display, but since the graphics display adapter card doesn't know how large a monitor is connected to it, it will appear smaller on smaller screens and larger on larger screens.)

The presentation space can also be set to what are called *arbitrary units,* using the PU_ARBITRARY identifier to GpiCreatePS. Arbitrary units can represent units in any measurement system. They are useful for units too large or small to be represented directly on the screen, such as miles and kilometers (used for drawing maps), and microns (used to measure electron microscope specimens).

When arbitrary units are used, the presentation page must be given appropriate nonzero dimensions, using the *sizl* argument to GpiCreatePS. You might specify a space 8000 kilometers long and 4000 kilometers high to hold a map of the United States, for example. The bottom left corner of the presentation page is always located at the origin of the presentation space.

If you use arbitrary units for your presentation space, you will usually need to perform a transformation to scale the picture appropriately for output. The device transformation using GpiSetPageViewport, as shown in the VIEWPORT example, is appropriate here.

CLIPPING

We've discussed how transformations operate between each coordinate space and the next. There is another mechanism at work between coordinate spaces: *clipping*. Before a picture is transformed, it can be clipped, so that only part of the picture undergoes the transformation into the next space. Clipping specifies a shape—usually a rectangle—in the first coordinate space. Everything in the rectangle will be copied to the next coordinate space, but everything outside will be discarded.

Types of Clipping

Like transformations, clipping can be carried out between world and model space, between model and page space, and between page and device space. This is shown in Figure 13-11.

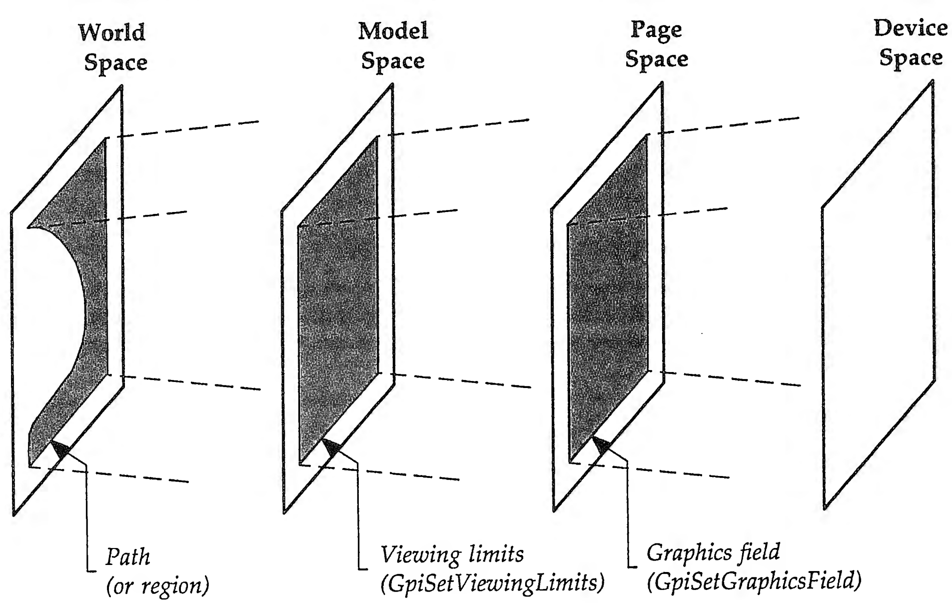


Figure 13-11. Clipping between coordinate spaces

Clipping makes it possible to transform only part of one space into the next space. For instance, suppose you have a picture of a house in one coordinate space. You might want to take only the east wing of the house and combine it with trees from another segment. So you clip the house to remove everything but the east wing.

In Chapter 12 we discussed how regions and paths can be used for clipping. Although we didn't mention it then, such clipping actually takes place between world and model space. This is the only place that a shape other than a rectangle can be used for clipping. A region can consist of a number of rectangles, and a path can have an arbitrary shape constructed from arcs and lines.

In model space a rectangle called the *viewing limits* can be defined, using the GpiSetViewingLimits function. Initially the viewing limits are the same size as the model space, so no clipping occurs. If the viewing limits are redefined using GpiSetViewingLimits, then whatever lies inside the viewing limits rectangle will be copied to page space, while anything outside will be clipped.

Similarly, in page space a rectangle called the *graphics field* can be defined, using the GpiSetGraphicsField function. Initially the graphics field is the same size as the page space, so no clipping occurs. If the graphics field is redefined with GpiSetGraphicsField, then whatever lies inside the graphics field rectangle will be copied to device space, while anything outside will be clipped.

The graphics field and viewing limits rectangles are not subject to the usual translations from model to page and from page to device space. Once you specify these rectangles, they will remain unaltered no matter what transformations are in effect.

Clipping and Printing

Our next example program demonstrates clipping, using a graphics field. It also shows some variations on how to print a graphics object and demonstrates some of the properties of presentation spaces.

The display output from the program is similar to that in the SCENE and VIEWPORT examples, except that a rectangle is drawn in the lower part of the window, as shown in Figure 13-12.

There is also an additional menu item, "Print". When this item is selected, whatever part of the scene is in the rectangle will be sent to the printer, as shown in Figure 13-13.

The listings for PRTSCENE.C, PRTSCENE.H, PRTSCENE, PRTSCENE.DEF, and PRTSCENE.RC follow.

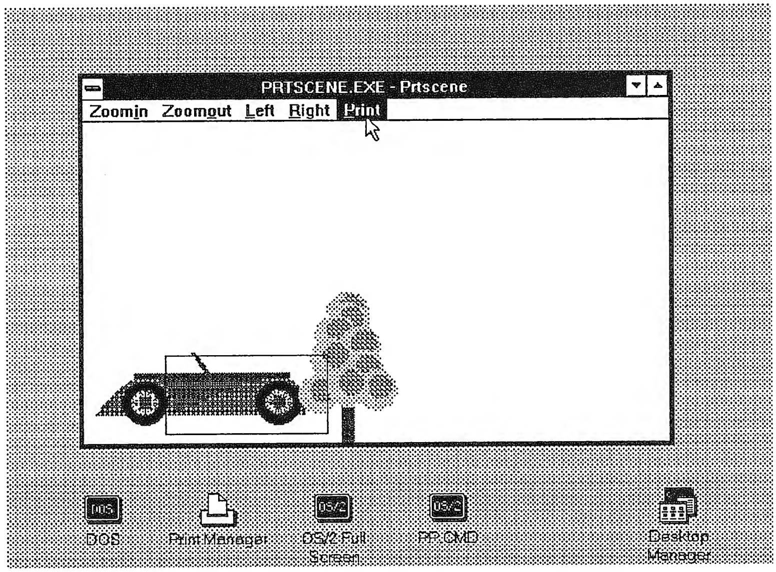


Figure 13-12. Display output of the PRTSCENE program

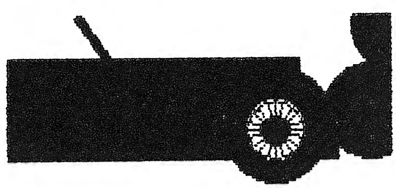


Figure 13-13. Printer output of the PRTSCENE program

```
/* ------------------------ */
/* PRTSCENE.C Print a scene */
/* ------------------------ */

#define INCL_GPI
#define INCL_WIN
#define INCL_DEV
#include <os2.h>
#include <string.h>

#include "prtscene.h"

HAB hab;                              /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                           /* handle for message queue */
   QMSG qmsg;                         /* message queue element */
   HWND hwnd, hwndClient;             /* handles for windows */

                                      /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_MENU;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);         /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);   /* create message queue */

                                      /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                      /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Prtscene", 0L, NULL, ID_FRAMERC,
             &hwndClient);

                                      /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);            /* destroy frame window */
   WinDestroyMsgQueue(hmq);           /* destroy message queue */
   WinTerminate(hab);                 /* terminate PM usage */

   return 0;
   }

                                      /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   static HDC hdcWin;
   HDC hdcPrt;
   static HPS hps;
   static SIZEL sizl = {0, 0};

   static POINTL ptl;
   int i;
```

```
static POINTL ptlFenders[6] = {{145, 230}, {220, 160}, {220, 70},
                               {450, 180}, {550, 210}, {545, 70}};

static POINTL ptlWind[3] = {{265, 225}, {275, 225}, {310, 175}};

static MATRIXLF matlfWheel2 = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                               MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                               330L, 0L, 1L};

static MATRIXLF matlfTree = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                             MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                             500L};

static POINTL branch[BRANCHES] = {{150,330}, {150,300},
                 {120,270}, {180,240}, {110,210}, {190,180},
                 {80,120}, {220,130}, {150,140}};

static MATRIXLF matlfZoomIn = {MAKEFIXED(2, 0), MAKEFIXED(0, 0), 0L,
                               MAKEFIXED(0, 0), MAKEFIXED(2, 0)};

static MATRIXLF matlfZoomOut =
                             {MAKEFIXED(0, 0x8000), MAKEFIXED(0, 0), 0L,
                              MAKEFIXED(0, 0), MAKEFIXED(0, 0x8000)};

static MATRIXLF matlfLeft = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                             MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                             -100L};

static MATRIXLF matlfRight = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                              MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                              100L};

static MATRIXLF matlfInitView =
                             {MAKEFIXED(0, 0x2000), MAKEFIXED(0, 0), 0L,
                              MAKEFIXED(0, 0), MAKEFIXED(0, 0x2000)};

MATRIXLF matlfView;

static DEVOPENSTRUC dop;
       CHAR szPrinter[32];
static CHAR szDetails[256];
USHORT cb;
static RECTL rclGraphicsField = {100L, 10L, 300L, 110L};
static POINTL ptlBox[] = {{100, 10}, {300, 110}};
RECTL rclTemp;

switch (msg)
   {
   case WM_PAINT:
      hps = WinBeginPaint(hwnd, hps, NULL);
      GpiErase(hps);

      GpiDrawChain(hps);              /* draw segment chain */
                                      /* save current default view */
      GpiQueryDefaultViewMatrix(hps, 7L, &matlfView);
      GpiResetPS(hps, GRES_ATTRS);    /* reset PS */
                                      /* set to unity view */
      GpiSetDefaultViewMatrix(hps, 0L, NULL, TRANSFORM_REPLACE);
                                      /* draw box */
```

```
      GpiMove(hps, &ptlBox[0]);
      GpiBox(hps, DRO_OUTLINE, &ptlBox[1], 0L, 0L);
                                      /* restore default view */
      GpiSetDefaultViewMatrix(hps, 7L, &matlfView, TRANSFORM_REPLACE);

      WinEndPaint(hps);
      break;

   case WM_COMMAND:
      switch (COMMANDMSG(&msg) -> cmd)
         {
         case ID_ZOOMIN:
            GpiSetDefaultViewMatrix(hps, 5L, &matlfZoomIn,
               TRANSFORM_ADD);
            break;
         case ID_ZOOMOUT:
            GpiSetDefaultViewMatrix(hps, 5L, &matlfZoomOut,
               TRANSFORM_ADD);
            break;
         case ID_LEFT:
            GpiSetDefaultViewMatrix(hps, 7L, &matlfLeft, TRANSFORM_ADD);
            break;
         case ID_RIGHT:
            GpiSetDefaultViewMatrix(hps, 7L, &matlfRight,
               TRANSFORM_ADD);
            break;
         case ID_PRINT:                /* print image in box */
                                       /* open printer DC */
            hdcPrt = DevOpenDC(hab, OD_QUEUED, "*", 4L,
                        (PDEVOPENDATA)&dop, (HDC)NULL);

            GpiAssociate(hps, NULL);      /* disassociate PS from win */
            GpiAssociate(hps, hdcPrt);    /* associate PS with prt DC */

                                          /* save old graphics field */
            GpiQueryGraphicsField(hps, &rclTemp);
                                       /* set graphics field (as box) */
            GpiSetGraphicsField(hps, &rclGraphicsField);

            GpiDrawChain(hps);         /* draw segment chain */

                                       /* restore graphics field */
            GpiSetGraphicsField(hps, &rclTemp);

            GpiAssociate(hps, NULL);      /* disassociate PS from prt */
            GpiAssociate(hps, hdcWin);    /* associate with window DC */
            DevCloseDC(hdcPrt);           /* close printer DC */
            break;
         }
      WinInvalidateRect(hwnd, NULL, FALSE);
      break;

   case WM_CREATE:
      hdcWin = WinOpenWindowDC(hwnd);
      hps = GpiCreatePS(hab, hdcWin, &sizl, PU_LOENGLISH |
               GPIT_NORMAL | GPIA_ASSOC);

      GpiSetDrawingMode(hps, DM_RETAIN);  /* set drawing mode */
```

```
GpiOpenSegment(hps, 0L);              /* open segment */
GpiSetColor(hps, CLR_RED);            /* draw body rectangle */
ptl.x = 125; ptl.y = 70;
GpiMove(hps, &ptl);
ptl.x = 505; ptl.y = 175;
GpiBox(hps, DRO_OUTLINEFILL, &ptl, 0L, 0L);

ptl.x = 300; ptl.y = 175;      /* draw windshield */
GpiBeginArea(hps, BA_BOUNDARY);
GpiMove(hps, &ptl);
GpiPolyLine(hps, 3L, ptlWind);
GpiEndArea(hps);

ptl.x = 30; ptl.y = 70;        /* draw fenders */
GpiMove(hps, &ptl);
GpiSetColor(hps, CLR_DARKRED);
GpiBeginArea(hps, BA_BOUNDARY);
GpiPolySpline(hps, 6L, ptlFenders);
GpiEndArea(hps);

GpiCallSegmentMatrix(hps, ID_WHEEL_SEG, 0L, NULL,
   TRANSFORM_REPLACE);
GpiCallSegmentMatrix(hps, ID_WHEEL_SEG, 7L. &matlfWheel2,
   TRANSFORM_REPLACE);

GpiCloseSegment(hps);          /* close segment */

                               /* create wheel segment */
GpiOpenSegment(hps, ID_WHEEL_SEG);   /* open segment */
ptl.x = 150; ptl.y = 100;
GpiMove(hps, &ptl);
GpiSetColor(hps, CLR_BLACK);         /* outer tire */
GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(55,0));
GpiSetColor(hps, CLR_BACKGROUND);    /* spoke background */
GpiFullArc(hps, DRO_FILL, MAKEFIXED(30,0));
GpiSetColor(hps, CLR_RED);
for(i = 0; i < 20; i++)              /* inner tire and spokes */
   {
   GpiMove(hps, &ptl);
   GpiPartialArc(hps, &ptl, MAKEFIXED(30,0),
      MAKEFIXED(i * 18,0), MAKEFIXED(18,0));
   }
GpiMove(hps, &ptl);            /* hubcap */
GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(15, 0));
GpiCloseSegment(hps);          /* close segment */

GpiSetSegmentAttrs(hps, ID_WHEEL_SEG, ATTR_CHAINED, ATTR_OFF);

GpiSetViewingTransformMatrix(hps, 7L, &matlfTree,
   TRANSFORM_REPLACE);

                               /* create tree */
GpiOpenSegment(hps, ID_TREE_SEG);    /* open segment */

ptl.x = 135; ptl.y = 0;              /* draw trunk */
GpiMove(hps, &ptl);
ptl.x = 165; ptl.y = 100;
GpiSetColor(hps, CLR_BROWN);
GpiBox(hps, DRO_FILL, &ptl, 0L, 0L);

for(i = 0; i < BRANCHES; i++)        /* draw branches */
```

```
        {
        GpiMove(hps, &branch[i]);
        GpiSetColor(hps, CLR_DARKGREEN);
        GpiFullArc(hps, DRO_FILL, MAKEFIXED(50,0));
        GpiSetColor(hps, CLR_GREEN);
        branch[i].x +=10; branch[i].y += 10;
        GpiMove(hps, &branch[i]);
        GpiFullArc(hps, DRO_FILL, MAKEFIXED(30,0));
        }
     GpiCloseSegment(hps);              /* close segment */

     GpiSetDrawingMode(hps, DM_DRAW);    /* set drawing mode */

     GpiSetDefaultViewMatrix(hps, 5L, &matlfInitView,
        TRANSFORM_REPLACE);

                                    /* get printer name */
     cb = WinQueryProfileString(hab, "PM_SPOOLER", "PRINTER", "",
            szPrinter, sizeof(szPrinter));
     szPrinter[cb - 2] = 0;         /* remove ";" */

     cb = WinQueryProfileString(hab, "PM_SPOOLER_PRINTER",
            szPrinter, "", szDetails, sizeof(szDetails));
     strtok(szDetails, ";");
     dop.pszDriverName = strtok(NULL, ";");
     dop.pszLogAddress = strtok(NULL, ";");
     strtok(dop.pszDriverName, ",");
     dop.pdriv = NULL;
     dop.pszDataType = "PM_Q_STD";
     break;

  case WM_DESTROY:
     GpiDeleteSegments(hps, ID_WHEEL_SEG, ID_TREE_SEG);
     GpiAssociate(hps, NULL);
     GpiDestroyPS(hps);
     break;

  default:
     return(WinDefWindowProc(hwnd, msg, mp1, mp2));
     break;
  }

return NULL;                        /* NULL / FALSE */
}
```

```
/* ---------- */
/* PRTSCENE.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_ZOOMIN 101
#define ID_ZOOMOUT 102
#define ID_LEFT 103
#define ID_RIGHT 104
#define ID_PRINT 110
```

```
#define ID_WHEEL_SEG 200L
#define ID_TREE_SEG 300L

#define BRANCHES 9
```

```
# ------------------
# PRTSCENE Make file
# ------------------

prtscene.obj: prtscene.c prtscene.h
   cl -c -G2s -W3 -Zp prtscene.c

prtscene.res: prtscene.rc prtscene.h
   rc -r prtscene.rc

prtscene.exe: prtscene.obj prtscene.def
   link /NOD prtscene,,NUL,os2 slibce,prtscene
   rc prtscene.res

prtscene.exe: prtscene.res
   rc prtscene.res
```

```
; ------------
; PRTSCENE.DEF
; ------------

NAME PRTSCENE WINDOWAPI

DESCRIPTION 'Print a scene'

PROTMODE
STACKSIZE   4096
```

```
/* ----------- */
/* PRTSCENE.RC */
/* ----------- */

#include <os2.h>
#include "prtscene.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "Zoom~in", ID_ZOOMIN
      MENUITEM "Zoom~out", ID_ZOOMOUT
      MENUITEM "~Left", ID_LEFT
      MENUITEM "~Right", ID_RIGHT
      MENUITEM "~Print", ID_PRINT
   END
```

Most of these listings are similar to the SCENE example. The picture is constructed from segments for the wheel, car body, and tree, as it was before. It can be zoomed and scrolled.

Printing

When the user selects the "Print" option from the menu bar, the part of the scene inside the rectangle on the screen will be printed; the scene is clipped at the borders of the rectangle.

Printing is carried out in much the same way as in the PRTGRAPH example in Chapter 10. However, there are a few refinements. One is that the code that does the printing is divided into two parts. Setting up the printer—the laborious process of finding out the driver name and logical address—is carried out during WM_CREATE processing. This means it will only be done once, when the program is started.

The actual printing is carried out when an ID_PRINT command is received with a WM_COMMAND message, as a result of the user selecting "Print" from the menu. Since we have already associated our presentation space with a window device context, when we're ready to print we must obtain a printer device context with DevOpenDC, disassociate the PS from the window device context, and reassociate it with the printer device context. At this point we can draw the picture chain. When printing is completed, we disassociate the PS from the printer device context using GpiAssociate with the second parameter set to NULL, and reassociate it with the window device context.

Clipping to a Graphics Field

Only that part of the image that's in the graphics field will be printed. We set the graphics field with the GpiSetGraphicsField function.

Set Graphics Field Coordinates

```
BOOL GpiSetGraphicsField(hps, prclField)
HPS hps             Handle to presentation space
PRECTL prclField    Graphics field rectangle

Returns: GPI_OK if successful, GPI_ERROR if error
```

The RECTL rectangle that defines the graphics field—the variable *rclGraphicsField* in our example—is initialized to fixed coordinates. This rectangle always occupies the same place in the program's window, no matter how much the user zooms and scrolls the scene, and no matter what size the window is. The coordinates of this rectangle are used not only to set the graphics field, but also to specify the outline of the rectangle that appears on the screen.

When printing is complete, we want to restore the previous graphics field so the screen will appear as before. This involves saving the existing graphics field before we set its new value. To find the existing graphics field, we use GpiQueryGraphicsField.

Retrieve Graphics Field Coordinates

```
BOOL GpiQueryGraphicsField(hps, prclField)
HPS hps          Handle to presentation space
PRECTL prclField Graphics field rectangle

Returns: GPI_OK if successful, GPI_ERROR if error
```

This function writes the coordinates for the existing graphics field into the structure of type RECTL pointed to by the second argument. When the picture chain has been drawn to the printer, the old graphics field is restored with GpiSetGraphicsField.

Keeping the Box Invariant

When the user zooms and scrolls the scene, the rectangle that outlines the graphics field does not change. How is this accomplished?

Before the rectangle is drawn, the PS is reset and a different transformation is applied than that used for the rest of the scene. Under WM_PAINT, as soon as WinBeginPaint is executed, the entire picture chain is drawn with GpiDrawChain. The scene that is drawn by the chain has all the attributes, such as color, set by functions in the picture chain. It also reflects all the zooming and scrolling selections made by the user and incorporated into various transformations. To draw the box, we want to use a different set of attributes and a different transformation.

The simplest way to reset all the attributes, so we can start over with a clean slate, is to reset the entire PS. For this we use the GpiResetPS function.

Reset Presentation Space

```
BOOL GpiResetPS(hps, flOption)
HPS hps          Handle to presentation space
ULONG flOption   Reset option

Returns: GPI_OK if successful, GPI_ERROR if error
```

The reset option can be one of the following:

Identifier	Resets or Deletes
GRES_ATTRS	Attributes, model transform, current position, path, area and element brackets; segments, clip path, and viewing limits
GRES_SEGMENTS	All of the above, plus retained segments, boundary data, clip region, segment attributes, default viewing transform, graphics field, drawing mode, draw controls, edit mode, and attribute mode. Enables kerning
GRES_ALL	All of the above, plus logical fonts, bitmap identifiers, and logical color table

We want to get rid of attributes without losing the retained segments that have already been defined, so we use the GRES_ATTRS identifier. With this option the default viewing transform is left in place.

We want to reset this transform to unity to draw the box, but the existing values must not be lost, since they incorporate the scrolling and zooming selections already made by the user. These existing values are therefore saved by using the GpiQueryDefaultViewMatrix function.

Retrieve Current Default Viewing Transformation

```
BOOL GpiQueryDefaultViewMatrix(hps, cElements, pmatlf)
HPS hps             Handle to presentation space
LONG cElements      Number of matrix elements
PMATRIXLF pmatlf    Address of matrix

Returns: GPI_OK if successful, GPI_ERROR if error
```

The function records *cElements* of the current default viewing matrix, storing them in a structure of type MATRIXLF. The unity (identity) matrix is then set with GpiSetDefaultViewMatrix.

Now the rectangle can be drawn into a PS that is free of preexisting attributes and transforms, using GpiMove and GpiBox. The rectangle will always appear on the screen in the same place, and in the default color, black. After the rectangle is drawn, the old default viewing transformation can be set again with GpiSetDefaultViewMatrix.

THINKING ABOUT COORDINATE SPACES

Understanding coordinate spaces, and the transformations and clipping that take place between them, can be confusing. To clarify the situation (or possibly to make it even murkier), consider these ideas.

No Physical Reality

The various coordinate spaces we talked about—world, model, page, and device—don't really exist. There is no place in memory (or anywhere else) where a picture is formed in these spaces. The picture has no existence until it reaches the physical output device. Our earlier metaphor of a coordinate space as a piece of paper is misleading in this respect. The concept of coordinate space is *only* a concept—one contrived to simplify thinking about transformations and clipping.

When a graphics primitive such as a line is drawn in world space, the points on the line are transformed from world space, to model space, to page space, to device space, and finally appear on the physical device itself. You start with a point, call it (10, 20), that you've specified in your program in world coordinates. It's drawn in world space. Then a transformation is applied to the point, yielding a corresponding point in model space, say (15, 23). Next a viewing transformation is applied, giving a point in page space, perhaps (17, 33). Then the device transformation is applied yielding a point in device space: (59, 42). Finally the point appears on the output device. But, except for the final step of drawing to the output device, this entire process is *algebraic.* The point doesn't really appear in any of the spaces, since the spaces don't exist. Instead PM is figuring out, from all the transforms in effect at that moment, where the point should appear on the output device.

Similarly, at each step from one space to another, only part of the scene is transformed; the rest is clipped off and discarded. But this doesn't mean that the part of the picture that was clipped—or the part that wasn't clipped—actually existed. PM is simply figuring out, from all the clipping rectangles in effect, whether a particular point should be displayed or not.

Only One Transformation

In fact, PM combines all the transforms into one big transformation, so it doesn't have to keep doing the arithmetic over and over. While we as programmers imagine three or more transforms being in effect, on a physical level only one is used. It transforms the coordinates we originally gave in world space to those appropriate for the output device—screen, printer, or whatever—that we're drawing to. Similarly, the clipping rectangles and the clipping path are combined into one clipping path to simplify PM's calculation of what will be displayed.

Only Control Points Transformed

While we may have given the impression that each point on a line, for example, is subject to all the transforms and clipping calculations, in reality only the *control points* for the line are transformed. For a line drawn by GpiLine, for example, only the current position and the endpoint of the line are transformed. The intermediate points on the line are calculated only for the output device, not for any of the intermediate spaces.

Thus, a line not only doesn't exist in any of the coordinate spaces, since they don't exist, but it also doesn't exist anywhere—except as the specifications for two points—until it actually appears on the output device.

Segments and Transformations

A segment can be thought of as occupying its own private world space. You can draw one object in one place in world coordinates and put it in segment A, and then draw another object and put it in the same place in world coordinates in segment B. These objects don't overlap, since they occupy different spaces. Of course in reality a segment doesn't exist in any space; it's simply a number of graphics orders that will draw a graphics object when invoked with GpiDrawChain or a similar API. When it's drawn, the relevant transformations and clipping will be applied.

In the next chapter we'll apply the concepts introduced here to more advanced topics.

14

ANIMATION, HIT TESTING, AND METAFILES

This chapter is concerned with three advanced graphics topics: animation, hit testing, and metafiles. Animation is carried out using a special kind of graphics segment—a dynamic segment—which can be redrawn in different places on the screen. Hit testing determines whether a graphic object falls within a defined area. A common use for hit testing is allowing the user to select objects on the screen by using the mouse pointer. Metafiles are a way to store and transfer graphics pictures. Among other uses, metafiles can be saved as disk files and transferred using the clipboard.

Although these three topics describe PM features that have different purposes, they are all related to the ideas of segments and retained graphics discussed in the preceding chapter.

ANIMATION

Animation involves causing a graphics object to appear to change its position on the screen with a smooth, continuous motion. It mimics the motion of objects in real life, such as a car driving along a road or a ball rolling on a billiard table. The program actually moves the object in a series of small steps, by removing the object from its old position and redrawing it in the new one. The background is left undisturbed by the passage of the animated object.

There are many ways to animate graphics objects in computer systems, but in PM, animation is most conveniently carried out using *dynamic segments.* A dynamic segment is similar to a normal segment, in that it is created the same way and may be part of the picture chain. However, a dynamic segment can be manipulated in ways that normal segments cannot. A dynamic segment can be drawn without drawing any of the normal segments, and it can be removed from the picture without affecting any of the normal segments.

A graphics object to be animated is placed in a dynamic segment. To animate the object, repeatedly follow these steps: remove the segment, alter or move it—commonly using a transformation—and redraw it. (Besides using a transformation to alter the object, you might also change its attributes, such as color, or otherwise alter it by editing the dynamic segment.)

The secret of dynamic segments is that they are always drawn using the XOR mix mode. As we noted in Chapter 12, if you XOR an object onto a background, and then XOR it again, it vanishes, restoring the original backgound. Removing a dynamic segment merely involves drawing the segment again to remove the object. (There is a potential problem here with a colored background, which we'll discuss later.)

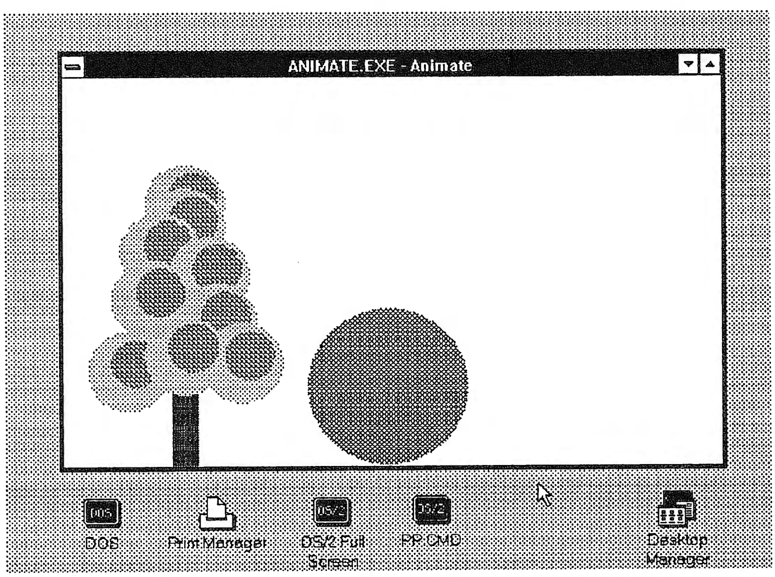


Figure 14-1. Output of the ANIMATE program

Our example program ANIMATE animates a circle. This circle represents a ball that appears to roll back and forth in front of a tree. Figure 14-1 shows how this looks.

Here are the listings for ANIMATE.C, ANIMATE.H, ANIMATE, and ANIMATE.DEF:

```
/* -------------------------------------------- */
/* ANIMATE.C  Animate a ball in front of a tree */
/* -------------------------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "animate.h"

HAB hab;                                  /* handle for anchor block */


USHORT cdecl main(void)
   {
   HMQ hmq;                               /* handle for message queue */
   QMSG qmsg;                             /* message queue element */
   HWND hwnd, hwndClient;                 /* handles for windows */

                                          /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);             /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);       /* create message queue */

                                          /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                          /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Animate", 0L, NULL, 0, &hwndClient);

                                          /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);                /* destroy frame window */
   WinDestroyMsgQueue(hmq);               /* destroy message queue */
   WinTerminate(hab);                     /* terminate PM usage */
   return 0;
   }

                                          /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   static HDC hdc;
   static HPS hps;
   static SIZEL sizl = {0, 0};

   POINTL ptl;
```

```
int i;

static POINTL branch[BRANCHES] = {{150,330}, {150,300},
                  {120,270}, {180,240}, {110,210}, {190,180},
                  {80,120}, {220,130}, {150,140}};

static MATRIXLF matlfMove = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                             MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                             STEP};
static USHORT Delay = 20;
static SHORT cMove = 0;

switch (msg)
   {
   case WM_PAINT:
      hps = WinBeginPaint(hwnd, hps, NULL);
      GpiErase(hps);

      GpiDrawChain(hps);              /* draw segment chain */

      GpiDrawDynamics(hps);           /* draw dynamic segments */

      WinEndPaint(hps);
      break;

   case WM_TIMER:
      if (SHORT1FROMMP(mp1) == TID_ANIMATE)
         {
         if (++cMove > CMAXMOVES)          /* if too far */
            {
            cMove = 0;                     /* change direction */
            matlfMove.lM31 = - matlfMove.lM31;
            }
                                           /* remove ball */
         GpiRemoveDynamics(hps, ID_BALL_SEG, ID_BALL_SEG);
                                           /* move ball */
         GpiSetSegmentTransformMatrix(hps, ID_BALL_SEG, 7L,
            &matlfMove, TRANSFORM_ADD);
         GpiDrawDynamics(hps);             /* redraw ball */
         }
         break;

   case WM_CREATE:
      hdc = WinOpenWindowDC(hwnd);
      hps = GpiCreatePS(hab, hdc, &sizl, PU_LOENGLISH |
               GPIT_NORMAL | GPIA_ASSOC);

      GpiSetDrawingMode(hps, DM_RETAIN);  /* set drawing mode */

      GpiSetInitialSegmentAttrs(hps, ATTR_DYNAMIC, ATTR_ON);

      GpiOpenSegment(hps, ID_BALL_SEG);    /* open segment */
      GpiSetColor(hps, CLR_RED);
      ptl.x = BALL_SIZE; ptl.y = BALL_SIZE;
      GpiMove(hps, &ptl);
      GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(BALL_SIZE, 0));
      GpiCloseSegment(hps);              /* close segment */

      GpiSetInitialSegmentAttrs(hps, ATTR_DYNAMIC, ATTR_OFF);

                                         /* create tree */
```

```
        GpiOpenSegment(hps, ID_-TREE_SEG); /* open segment */

        ptl.x = 135; ptl.y = 0;            /* draw trunk */
        GpiMove(hps, &ptl);
        ptl.x = 165; ptl.y = 100;
        GpiSetColor(hps, CLR_BROWN);
        GpiBox(hps, DRO_FILL, &ptl, 0L, 0L);

        for(i = 0; i < BRANCHES; i++)      /* draw branches */
           {
           GpiMove(hps, &branch[i]);
           GpiSetColor(hps, CLR_DARKGREEN);
           GpiFullArc(hps, DRO_FILL, MAKEFIXED(50,0));
           GpiSetColor(hps, CLR_GREEN);
           branch[i].x += 10; branch[i].y += 10;
           GpiMove(hps, &branch[i]);
           GpiFullArc(hps, DRO_FILL, MAKEFIXED(30,0));
           }

        GpiCloseSegment(hps);              /* close segment */

        GpiSetDrawingMode(hps, DM_DRAW);   /* set drawing mode */

        WinStartTimer(hab, hwnd, TID_ANIMATE, Delay);
        break;

     case WM_DESTROY:
        GpiDeleteSegments(hps, ID_TREE_SEG, ID_BALL_SEG);
        GpiAssociate(hps, NULL);
        GpiDestroyPS(hps);
        break;

     default:
        return(WinDefWindowProc(hwnd, msg, mp1, mp2));
        break;
     }

  return NULL;                         /* NULL / FALSE */
  }
```

```
/* --------- */
/* ANIMATE.H */
/* --------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_TREE_SEG 300L
#define ID_BALL_SEG 400L

#define BRANCHES 9

#define TID_ANIMATE 800

#define BALL_SIZE 100
#define CMAXMOVES 70
#define STEP 10L
```

```
# ------------------
# ANIMATE Make file
# ------------------

animate.obj: animate.c animate.h
   cl -c -G2s -W3 -Zp animate.c

animate.exe: animate.obj animate.def
   link /NOD animate,,NUL,os2 slibce,animate
```

```
; -----------
; ANIMATE.DEF
; -----------

NAME   ANIMATE WINDOWAPI

DESCRIPTION 'Animate'

PROTMODE
STACKSIZE   4096
```

There are two kinds of segments in this program. The tree is drawn as a normal segment with GpiDrawChain. The ball is drawn as a dynamic segment with GpiDrawDynamics. Note that GpiDrawChain doesn't draw the dynamic segments, and GpiDrawDynamics doesn't draw the normal segments. Thus, we can draw a background (the tree) once with GpiChain, and then repeatedly draw the dynamic segment (the ball) with GpiDrawDynamics without disturbing the background.

Creating a Dynamic Segment

The dynamic segment containing the ball is created during WM_CREATE processing. To give it the dynamic attribute, the GpiSetInitialSegmentAttrs function is executed just before the segment is opened. This function is similar to GpiSetSegmentAttrs, described in the previous chapter, except that it sets an attribute for all segments opened subsequently, while GpiSetSegmentAttrs sets an attribute for a particular specified segment. Either function could be used here.

Set Attributes for Subsequent Segments

```
BOOL GpiSetInitialSegmentAttrs(hps, flAttribute, flAttrFlag)
HPS hps              Handle to presentation space
LONG flAttribute     Attribute to be changed: ATTR_CHAINED, etc.
LONG flAttrFlag      On/off flag: ATTR_ON or ATTR_OFF

Returns: GPI_OK if successful, GPI_ERROR if error
```

This function can change the same attributes as those listed for GpiSetSegmentAttrs. The attribute we're interested in has an identifier of ATTR_DYNAMIC, so we turn this on with ATTR_ON, define the segment, and then execute GpiSetInitialSegmentAttrs again with ATTR_OFF to return to normal segments. After this we define the segment that draws the tree, which is not dynamic, since the tree doesn't move.

Timers and Animation

The last act the program commits during WM_CREATE processing is starting a timer. What's this used for? You might imagine that you could perform animation in a loop, alternately removing the object and redrawing it in different places. Remember, however, that a PM application must be ready to respond to messages in a fraction of a second. Continuous loops are inappropriate in such an architecture (unless a separate thread is used for message processing). To avoid this problem, the timer is used to specify how often our graphics object will be moved. The timer is started with WinStartTimer.

We use the timer message WM_TIMER to run the animation. Each time this message is received, the existing dynamic segment is removed, the transform that positions it is changed, and it is redrawn in the new location.

Moving the Dynamic Segment

To move the dynamic segment, we first remove it from its existing position on the screen with GpiRemoveDynamics. When this function is executed, the graphics object drawn by the segment will vanish. This function simply draws the object in the same place with the XOR mix mode, which has the effect of removing the object.

Remove Dynamic Segments from Display

```
BOOL GpiRemoveDynamics(hps, idFirstSegment, idLastSegment)
HPS hps                 Handle to presentation space
LONG idFirstSegment     First segment in section to be removed
LONG idLastSegment      Last segment in section to be removed

Returns: GPI_OK if successful, GPI_ERROR if error
```

Arguments to this function allow you to remove (cause to disappear) a sequence of dynamic segments from the picture chain. To do this, you

specify the IDs of the first and last segments to be removed. In our example we remove only one segment, ID_BALL_SEG, so both *idFirstSegment* and *idLastSegment* are set to this ID.

To change the position of the segment, we execute GpiSetSegmentTransformMatrix. This function sets the transformation between world and model space for a specific segment.

Set Segment Transformation

```
BOOL GpiSetSegmentTransformMatrix(hps, idSegment, cElements,
                                  pmatlf, flType)
HPS hps             Handle to presentation space
LONG idSegment      Segment ID
LONG cElements      Number of valid elements in matrix (0L to 9L)
PMATRIXLF pmatlf    Matrix
LONG flType         Transform type

Returns: GPI_OK if successful, GPI_ERROR if error
```

The arguments to this function are identical to those for GpiCallSegmentMatrix. However, that function is unique in that it draws the segment and changes the segment transformation for that single drawing operation. GpiSetSegmentTransformMatrix changes the transformation for all subsequent drawing operations, but does not draw the segment.

In addition to moving the ball, the segment transformation is responsible for changing its direction. This is the first act following receipt of the WM_TIMER message. If the ball has gone too far—either right or left—the *lM31* element of the *matlfMove* matrix, which controls translation in the X direction, undergoes a sign change.

Once the transformation is changed, we can draw the dynamic segment with GpiDrawDynamics.

Draw Dynamic Segments in Picture Chain

```
BOOL GpiDrawDynamics(hps)
HPS hps          Handle to presentation space

Returns: GPI_OK if successful, GPI_ERROR if error
```

This function draws all the dynamic segments in the chain. In our example there is only one, ID_BALL_SEG.

If there are multiple dynamic segments and only some of them have been removed with GpiRemoveDynamics, then GpiDrawDynamics is smart enough to redraw only those removed segments; segments that were not removed are not redrawn.

Animation and Color

Dynamic segments work well when the background is black. However, if you draw a dynamic segment on a colored background, you may run into trouble. The problem occurs because dynamic segments XOR the object with the background, as can be seen in the example. Unlike the overpaint mix mode, where the foreground object replaces the background colors, XORing the colors typically produces a mixture of the foreground and background colors. The resulting color depends on what RGB color values correspond to the color indices used in the table.

To animate a colored object on a colored background, you will probably need to modify the logical color table using GpiCreateLogicalColorTable. You will usually want to rearrange the table so that the colors that result from XORing particular object colors with particular background colors result in unchanged object colors. For example, you want a red car passing in front of a green tree to remain red. The exact arrangement to use in the color table depends on the colors used for the object and the background. We don't manipulate the color table in this example.

HIT TESTING

Hit testing is the process of finding out if a graphics object on the screen coincides with the mouse pointer, or with another graphics object. One common use for hit testing is to allow the user to select objects using the mouse. For example, in an architectural program the user might want to relocate the kitchen sink from one place in the kitchen to another. The program would use hit testing to determine that the user had selected the sink. It could then move the sink, using animation, as the user dragged it with the mouse.

Hit testing is also used to determine when one animated graphics object runs into another. In an auto racing program it might be used to determine when a car runs off the track, or into another car.

We've already seen several rudimentary kinds of hit testing. In Chapter 5, the POSITION example read the mouse coordinates from the WM_MOUSEMOVE message and beeped if the pointer was to the left of a line. The PUZZLE example in Chapter 9 determined which of 16 windows was being pointed at. And the REGION program in Chapter 12 determined if the pointer was in a particular region when the button was pressed. These methods all determine information about the mouse pointer in relation to a particular area on the screen, usually a rectangle.

In many cases, however, we want to perform hit testing on graphics objects with shapes more complex than a rectangle, such as cars and spaceships. Trying to determine by calculation if a car and another object were in the same place would be quite difficult, because the car is such a complicated shape. However, we can let PM figure it out, using a technique called correlation.

Correlation

Correlation is the process of finding out if a graphics object touches something called the *pick aperture*. The pick aperture is a rectangle whose position and size can be defined by the program. Once the pick aperture's size and position are determined, PM can then figure out what graphics objects intersect this rectangle. Actually, each primitive in the object—lines, arcs, areas, and so on—is checked separately. If a primitive intersects the pick aperture, a "hit" is said to have occurred.

The process that PM uses to figure out if the pick aperture intersects a graphics primitive may seem backwards at first. You might think that PM would draw the picture, then see where the pick aperture was, and finally calculate what graphics primitives the pick aperture touched. However, once a primitive is drawn on the screen, PM has no convenient way of knowing where it is. So it takes another approach. It determines the location of the pick aperture first, and then redraws the picture (or at least part of the picture). During the redrawing process, it checks each primitive to see if it coincides with the pick aperture. If it does, the program is informed of a hit. (Actually, this redrawing process does the calculations, but does not draw any pixels on the screen.)

The advantage of this approach is that any graphics object, no matter how complicated, can be correlated. The disadvantage is that all the graphics objects you might want to correlate on must be redrawn every time you want to check for a hit. This can take time.

The Pick Aperture

You set the position of the pick aperture using the GpiSetPickAperture Position function. By default the pick aperture is a square with sides equal to an average character height. You can change the dimensions of the rectangle with the GpiSetPickApertureSize function. A hit, or correlation, takes place when any graphics primitive falls within this rectangle.

If the pick aperture is too small, the user may find it requires too much care to position properly. Increasing the size of the pick aperture allows the user to be a little sloppier, and also permits several adjacent graphics objects to be selected at the same time, if that's desirable. The size of the pick aperture can be adjusted so it is large enough not to require too much precision from the user, but small enough not to select too many objects at once.

Correlating on Tags

In our example we'll demonstrate one approach to correlation. This approach is appropriate when the user will be using the mouse to select one of a number of objects that appear on the screen. The example also demonstrates how to draw the same segment in different places. It draws three cars and four trees, as shown in Figure 14-2.

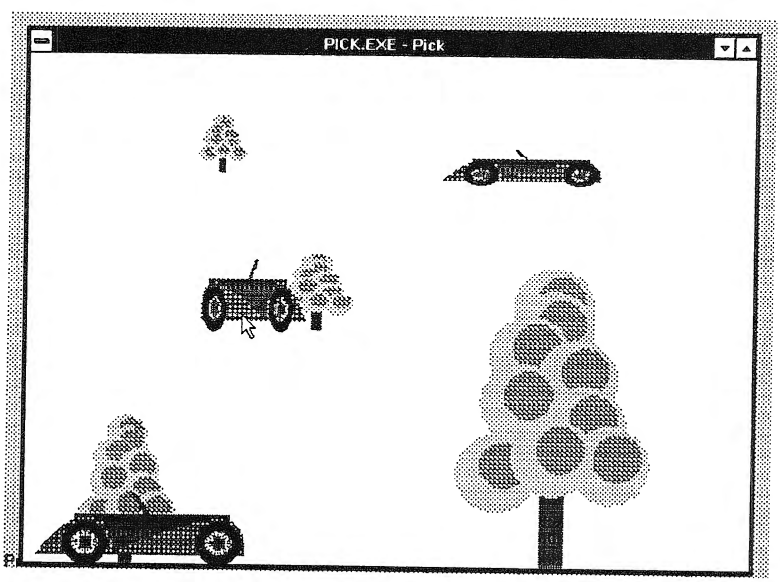


Figure 14-2. Output of the PICK program

When the user clicks on one of the cars, the program sounds a beep. ("Clicking on" means moving the mouse pointer to an object and pressing the left button.) There is a different beep for each car. Correlation is used to determine whether a car is under the pointer when the button is pressed, and if so, which car it is.

Here are the listings for PICK.C, PICK.H, PICK, and PICK.DEF:

```
/* ------------------ */
/* PICK.C  Pick a car */
/* ------------------ */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "pick.h"

HAB hab;                               /* handle for anchor block */


USHORT cdecl main(void)
   {
   HMQ hmq;                            /* handle for message queue */
   QMSG qmsg;                          /* message queue element */
   HWND hwnd, hwndClient;              /* handles for windows */

                                       /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);          /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);    /* create message queue */

                                       /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                       /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
            szClientClass, " - Pick", 0L, NULL, 0, &hwndClient);

                                       /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);             /* destroy frame window */
   WinDestroyMsgQueue(hmq);            /* destroy message queue */
   WinTerminate(hab);                  /* terminate PM usage */

   return 0;
   }

                                       /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   static HDC hdc;
```

```
static HPS hps;
static SIZEL sizl = {0, 0};

static POINTL ptl;
int i;

static POINTL ptlFenders[6] = {{145, 230}, {220, 160}, {220, 70},
                               {450, 180}, {550, 210}, {545, 70}};

static POINTL ptlWind[3] = {{265, 225}, {275, 225}, {310, 175}};

static MATRIXLF matlf = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                         MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                         330L, 0L, 1L};

static MATRIXLF matlfTrees[NTREES] =
   {{MAKEFIXED(0, 0x8000), MAKEFIXED(0, 0), 0L,
     MAKEFIXED(0, 0), MAKEFIXED(0, 0x8000), 0L,
     50L, 0L},
    {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
     MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
     500L, 0L},
    {MAKEFIXED(0, 0x6000), MAKEFIXED(0, 0), 0L,
     MAKEFIXED(0, 0), MAKEFIXED(0, 0x4000), 0L,
     300L, 300L},
    {MAKEFIXED(0, 0x4000), MAKEFIXED(0, 0), 0L,
     MAKEFIXED(0, 0), MAKEFIXED(0, 0x3000), 0L,
     200L, 500L}};

static MATRIXLF matlfCars[NCARS] =
   {{MAKEFIXED(0, 0x8000), MAKEFIXED(0, 0), 0L,
     MAKEFIXED(0, 0), MAKEFIXED(0, 0x8000), 0L,
     0L, -25L},
    {MAKEFIXED(-1, 0xC000), MAKEFIXED(0, 0), 0L,
     MAKEFIXED(0, 0), MAKEFIXED(0, 0x8000), 0L,
     350L, 275L},
    {MAKEFIXED(0, 0x6000), MAKEFIXED(0, 0), 0L,
     MAKEFIXED(0, 0), MAKEFIXED(0, 0x4000), 0L,
     500L, 475L}};

static POINTL branch[BRANCHES] = {{150,330}, {150,300},
                 {120,270}, {180,240}, {110,210}, {190,180},
                 {80,120}, {220,130}, {150,140}};

POINTL ptlAperture;
LONG alTagSel[2];                       /* for Segment/Tag of hits */

switch (msg)
   {
   case WM_BUTTON1DOWN:
      ptlAperture.x = SHORT1FROMMP(mp1);
      ptlAperture.y = SHORT2FROMMP(mp1);
      GpiConvert(hps, CVTC_DEVICE, CVTC_DEFAULTPAGE, 1L, &ptlAperture);

                                        /* if tagged */
      if (GpiCorrelateChain(hps, PICKSEL_VISIBLE, &ptlAperture, 1L, 1L,
         &alTagSel[0]) > 0)
                                        /* sound alarm */
         WinAlarm(HWND_DESKTOP, (SHORT)alTagSel[1] - 1);
      return TRUE;
      break;
```

```
case WM_PAINT:
   hps = WinBeginPaint(hwnd, hps, NULL);
   GpiErase(hps);

   GpiDrawChain(hps);              /* draw segment chain */

   WinEndPaint(hps);
   break;

case WM_CREATE:
   hdc = WinOpenWindowDC(hwnd);
   hps = GpiCreatePS(hab, hdc, &sizl, PU_LOENGLISH |
            GPIT_NORMAL | GPIA_ASSOC);

   GpiSetDrawingMode(hps, DM_RETAIN);  /* set drawing mode */

   GpiSetInitialSegmentAttrs(hps, ATTR_CHAINED, ATTR_OFF);

   GpiOpenSegment(hps, ID_CAR_SEG);    /* open segment */
   GpiSetColor(hps, CLR_RED);          /* draw body rectangle */
   ptl.x = 125; ptl.y = 70;
   GpiMove(hps, &ptl);
   ptl.x = 505; ptl.y = 175;
   GpiBox(hps, DRO_OUTLINEFILL, &ptl, 0L, 0L);

   ptl.x = 300; ptl.y = 175;       /* draw windshield */
   GpiBeginArea(hps, BA_BOUNDARY);
   GpiMove(hps, &ptl);
   GpiPolyLine(hps, 3L, ptlWind);
   GpiEndArea(hps);

   ptl.x = 30; ptl.y = 70;         /* draw fenders */
   GpiMove(hps, &ptl);
   GpiSetColor(hps, CLR_DARKRED);
   GpiBeginArea(hps, BA_BOUNDARY);
   GpiPolySpline(hps, 6L, ptlFenders);
   GpiEndArea(hps);

   GpiCallSegmentMatrix(hps, ID_WHEEL_SEG, 0L, NULL,
      TRANSFORM_PREEMPT);
   GpiCallSegmentMatrix(hps, ID_WHEEL_SEG, 7L, &matlf,
      TRANSFORM_PREEMPT);

   GpiCloseSegment(hps);               /* close segment */

                                   /* create wheel segment */
   GpiOpenSegment(hps, ID_WHEEL_SEG);           /* open segment */
   ptl.x = 150; ptl.y = 100;
   GpiMove(hps, &ptl);
   GpiSetColor(hps, CLR_BLACK);     /* outer tire */
   GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(55,0));
   GpiSetColor(hps, CLR_BACKGROUND); /* spoke background */
   GpiFullArc(hps, DRO_FILL, MAKEFIXED(30,0));
   GpiSetColor(hps, CLR_RED);
   for (i = 0; i < 20; i++)         /* inner tire and spokes */
      {
      GpiMove(hps, &ptl);
      GpiPartialArc(hps, &ptl, MAKEFIXED(30,0),
         MAKEFIXED(i*18,0), MAKEFIXED(18,0));
      }
   GpiMove(hps, &ptl);              /* hubcap */
```

```
    GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(15,0));
    GpiCloseSegment(hps);                /* close segment */

                                         /* create tree */
    GpiOpenSegment(hps, ID_TREE_SEG);  /* open segment */

    ptl.x = 135; ptl.y = 0;              /* draw trunk */
    GpiMove(hps, &ptl);
    ptl.x = 165; ptl.y = 100;
    GpiSetColor(hps, CLR_BROWN);
    GpiBox(hps, DRO_FILL, &ptl, 0L, 0L);

    for (i = 0; i < BRANCHES; i++)       /* draw branches */
       {
       GpiMove(hps, &branch[i]);
       GpiSetColor(hps, CLR_DARKGREEN);
       GpiFullArc(hps, DRO_FILL, MAKEFIXED(50,0));
       GpiSetColor(hps, CLR_GREEN);
       branch[i].x += 10; branch[i].y += 10;
       GpiMove(hps, &branch[i]);
       GpiFullArc(hps, DRO_FILL, MAKEFIXED(30,0));
       }

    GpiCloseSegment(hps);           /* close segment */

    GpiSetInitialSegmentAttrs(hps, ATTR_CHAINED, ATTR_ON);

    GpiOpenSegment(hps, ID_TREES_SEG);

                                    /* draw trees */
    for (i = 0; i < NTREES; i++)
       GpiCallSegmentMatrix(hps, ID_TREE_SEG, 8L, &matlfTrees[i],
          TRANSFORM_REPLACE);

    GpiCloseSegment(hps);


    GpiOpenSegment(hps, ID_CARS_SEG);
                                    /* draw cars */
    for (i = 0; i < NCARS; i++)
       {
       GpiSetTag(hps, (LONG)i+1);
       GpiCallSegmentMatrix(hps, ID_CAR_SEG, 8L, &matlfCars[i],
          TRANSFORM_ADD);
       }

    GpiCloseSegment(hps);
    GpiSetSegmentAttrs(hps, ID_CARS_SEG, ATTR_DETECTABLE, ATTR_ON);

    GpiSetDrawingMode(hps, DM_DRAW);  /* set drawing mode */

    break;

 case WM_DESTROY:
    GpiDeleteSegments(hps, ID_CAR_SEG, ID_CARS_SEG);
    GpiAssociate(hps, NULL);
    GpiDestroyPS(hps);
    break;

 default:
    return(WinDefWindowProc(hwnd, msg, mp1, mp2));
```

```
        break;
     }

   return NULL;                          /* NULL / FALSE */
   }
```

```
/* ------ */
/* PICK.H */
/* ------ */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_CAR_SEG 200L
#define ID_WHEEL_SEG 210L
#define ID_TREE_SEG 300L
#define ID_TREES_SEG 400L
#define ID_CARS_SEG 401L

#define BRANCHES 9

#define NTREES 4
#define NCARS 3
```

```
# --------------
# PICK Make file
# --------------

pick.obj: pick.c pick.h
   cl -c -G2s -W3 -Zp pick.c

pick.exe: pick.obj pick.def
   link /NOD pick,,NUL,os2 slibce,pick
```

```
; --------
; PICK.DEF
; --------

NAME        PICK WINDOWAPI

DESCRIPTION 'Pick a car'

PROTMODE
STACKSIZE   4096
```

Replicating a Segment

Before we examine how correlation is achieved in this program, let's see how multiple trees and cars are created. It's a two-step process. First, we define unchained segments for the car body, car wheel, and tree. The car

and tree are ID_CAR_SEG and ID_TREE_SEG, respectively. Then we define chained segments that call these unchained segments multiple times. The chained segments are ID_CARS_SEG and ID_TREES_SEG (note the plural names). For each call to an unchained segment from a chained segment, we change the transform to reposition the graphics object. This arrangement is shown in Figure 14-3.

To facilitate this approach, the transform matrices for the cars and for the trees are arranged in an array. When the chained segments call the unchained segments containing the car and the tree, they do so with GpiCallSegmentMatrix, using a matrix specified by an array element like *&matlfCars[i]*, where the index *i* changes for each car. Thus, each car can be transformed into a different size and a different position. One car is even reflected so it's facing the other way. Once all the segments have been defined, the detectability attribute of the ID_CARS_SEG is turned on with GpiSetSegmentAttrs. Segments are not detectable unless this attribute is specifically turned on. The result is that the user can select the cars, but not the trees.

Note that you can only perform correlation on named (numbered) segments. You can't correlate on segments opened with an ID of 0.

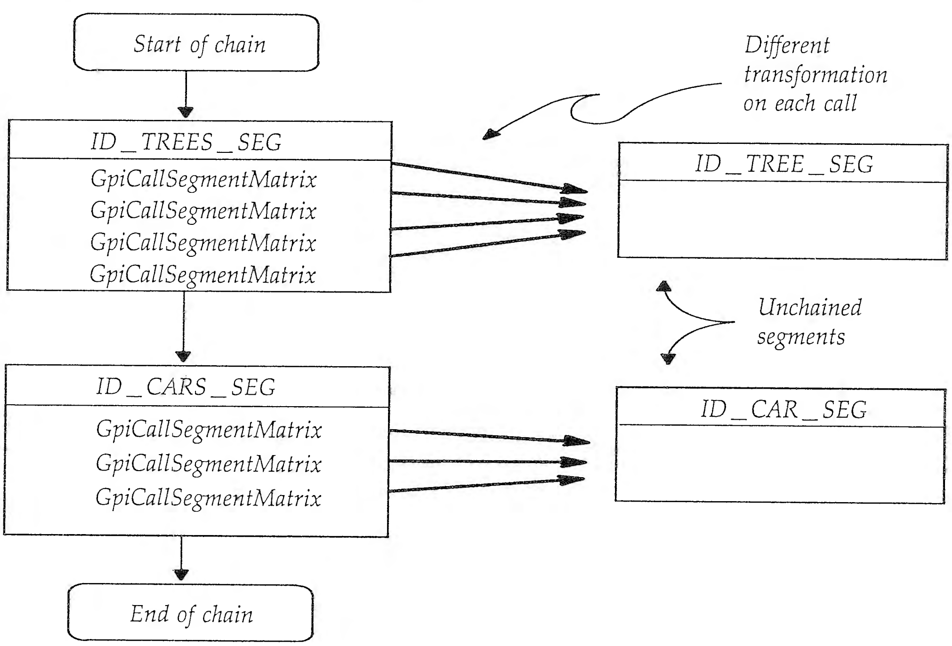


Figure 14-3. Calling unchained segments

Tagging Graphics Primitives

When we correlate on a graphics object, the Gpi correlation functions will return the ID of the segment that caused the hit. Thus, if we want to see if a car touched the pick aperture, we'd check whether the ID of the segment that drew the car was reported. However, sometimes we need more precise information than simply the segment: We also want to know exactly what primitive (or group of primitives) within the segment was responsible for the hit.

To return information about particular primitives, we need to give unique ID numbers to the primitives. These IDs are called *tags.* A tag can be applied to a single primitive or to a group of primitives.

To tag a group of primitives, we use GpiSetTag (a refreshingly brief API name). Once a tag number is specified by this function, the same tag is given to all primitives until the tag is changed by another call to GpiSetTag.

Set Current Primitive Tag

```
BOOL GpiSetTag(hps, lTag)
HPS hps            Handle to presentation space
LONG lTag          Tag (any integer value)

Returns: GPI_OK if successful, GPI_ERROR if error
```

In our example, GpiSetTag appears only in the loop that defines the cars. Our use of this function is slightly tricky. Remember that the three cars are drawn by calling the same segment, ID_CAR_SEG, three times. Thus, we can't distinguish the three cars by segment number. Instead, we tag all the primitives in the car with one number the first time we call the segment, another number the second time, and a third number the third time. So all the primitives for a given car have the same tag. The effect is that each car is given a unique tag. The trees, which will not be correlated on, are not given tags.

The GpiCorrelateChain Function

When the user pushes the mouse button, the application receives a WM_BUTTON1DOWN message. It finds the location of the mouse pointer from *mp1* and uses GpiConvert to convert it from pixels to default page coordinates (page coordinates before the default viewing transform has been applied). This point is stored in *ptlAperture.*

With this information we're ready to see if there is a hit, that is, if the pick aperture, centered on the mouse pointer, overlapped one of the cars at the time the button was pushed. We do this with the GpiCorrelateChain function. This function is given the center of the pick aperture. It then searches for hits, that is, primitives that lie within the pick aperture. It searches only named segments, and only tagged primitives within these segments. It does this by drawing the named segments and tagged primitives, and checking for correlations with each primitive as it draws it.

The results are returned in an array of type LONG. Each time any part of a primitive is drawn in the pick aperture, GpiCorrelateChain stores the ID of the segment containing the primitive, and the primitive tag, in the array. This sounds straightforward, but what happens if the segment being recorded was called by another segment, which was called by yet another segment, and so on? Our application may need the numbers of these ancestor segments. The function takes care of this by also inserting the identifiers of the ancestor segments into the array. We can specify how many generations of segments we want to record. This is called the *depth.* GpiCorrelateChain records the segment responsible for the hit, the segment that called that segment, and so on until it either reaches the root segment or exceeds the specified depth. Each ancestor segment is recorded with the same tag. If there are fewer segments to record than are specified by the depth, then the unused spaces in the array are filled in with zeros. Figure 14-4 shows such an array.

In the figure the depth is set to 3, so up to three generations of segment/tag pairs can be recorded. For example, the first hit was caused by a primitive (or group of primitives) with a tag of 17. This primitive is in segment 110, and this segment was called by segment 100, which is a root segment. The third hit was caused by a group of primitives with a tag of 33, which is in root segment 300.

Correlate Segment Chain

```
LONG GpiCorrelateChain(hps, lType, pptl, lMaxHits, lMaxDepth,
                       alSegTag)
HPS hps            Handle to presentation space
LONG lType         Type: PICKSEL_ALL or PICKSEL_VISIBLE
PPOINTL pptl       Center of pick aperture
LONG lMaxHits      Maximum number of hits to record
LONG lMaxDepth     Maximum number of segment/tag pairs to record
PLONG alSegTag     Array for segment/tag pairs

Returns: Number of hits if successful, GPI_ERROR if error
```

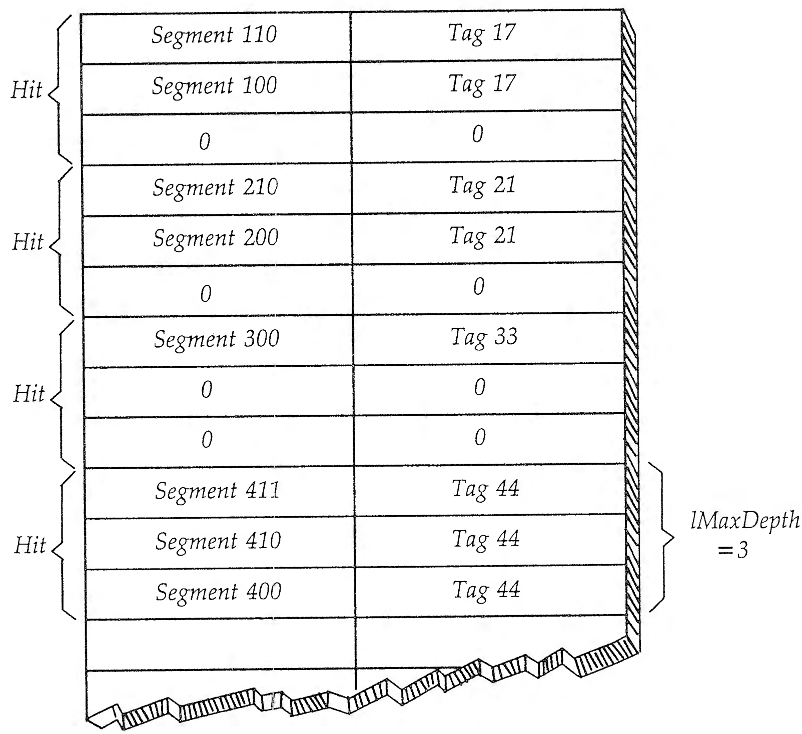


Figure 14-4. The segment/tag array

The *lType* argument to this function specifies whether segments that are not detectable and visible will be correlated.

Identifier	**Correlate**
PICKSEL_ALL	All segments with nonzero identifiers
PICKSEL_VISIBLE	Only visible and detectable segments

The *pptl* argument points to the center of the pick aperture (which in our example is set to the mouse pointer position). The *lMaxHits* argument tells how many hits the array can store, and *lMaxDepth* is the depth for each hit. The array *alSegTag* holds the segment/tag pairs. As can be seen from Figure 14-4, the size of this array is *lMaxDepth* * *lMaxHits* * 2.

In our program we don't need any information about the segments. All we need is the tag, since we've set up the tags to uniquely identify each car. Thus, in the segment/tag array we check only *alTagSel[1]*, which is the tag for the first hit. We assume there is never more than one hit when WM_BUTTON1DOWN is received, since the cars don't overlap. Thus both *lMaxHits* and *lMaxDepth* are set to 1L.

The number of the tag, which will be 1, 2, or 3, is used (with 1 subtracted) as an argument to WinAlarm to specify which of the three possible tones will sound.

Two other functions can be used for correlation. GpiCorrelateSegment specifies a particular segment, rather than correlating on the entire chain. GpiCorrelateFrom correlates on a section of the segment chain, from *idFirstSegment* to *idLastSegment*.

Correlation Using GPI_HITS

So far we've shown how correlation is performed using special APIs that fill arrays with segment/tag pairs representing hits. This approach is suited to the situation where many objects are drawn and you want to see which of them intersects the pick aperture, which is typically centered on the mouse pointer to allow the user to select one of a number of objects.

Another approach is appropriate when you want to check whether one graphics object coincides with another. This involves checking the return value from each graphics primitive. You do this in draw (non-retain) mode, when you're drawing an object directly to the screen using Gpi calls. Here are the necessary steps:

- Switch on the correlate flag with GpiSetDrawControl.
- Set the pick aperture position with GpiSetPickAperturePosition and size with GpiSetPickApertureSize.
- Draw the picture using Gpi calls like GpiLine and GpiBox.
- Check the return value of each Gpi function as it's drawn. If the value is GPI_HITS, that primitive intersected the pick aperture. If it's GPI_OK, no hit occurred.

You might use this approach to check if a meteor hit a spaceship. You would place the pick aperture on the ship, then draw the path of the meteor and check each of its primitives to see if GPI_HITS is returned.

METAFILES

Metafiles provide a way to store graphics images. They are used to store pictures in disk files, to transfer pictures to other applications using the clipboard, and in other situations.

Despite its name, a metafile does not start out as a disk file. Instead, it's stored in memory. From memory the contents of the metafile can be written to a disk file or processed in other ways.

A metafile contains all the information about a picture to be drawn. This picture is stored as a series of Gpi statements coded as graphics orders, just as it is in a retained segment. A metafile also contains various elements normally found in a presentation space, such as the logical color table, fill pattern, presentation page units and dimensions, and viewing transformations. All the output that you would normally send to a device such as the screen or a printer to create a picture is written to a metafile.

Our example consists of two programs, SAVEMETA and LOADMETA. SAVEMETA creates a metafile and draws into it the picture of the car and tree that we've seen in previous programs. It then saves this metafile to a disk file. LOADMETA reads the same disk file and "plays" the metafile, displaying it on the screen.

Here are the listings for the first program: SAVEMETA.C, SAVEMETA.H, SAVEMETA, SAVEMETA.DEF, and SAVEMETA.RC:

```
/* -------------------------- */
/* SAVEMETA.C Save a metafile */
/* -------------------------- */

#define INCL_GPI
#define INCL_WIN
#define INCL_DEV
#include <os2.h>
#include <string.h>

#include "savemeta.h"

HAB hab;                                /* handle for anchor block */


USHORT cdecl main(void)
   {
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd, hwndClient;               /* handles for windows */
```

```
                                             /* create flags */
    ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                          FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                          FCF_MENU;

    static CHAR szClientClass[] = "Client Window";

    hab = WinInitialize(NULL);               /* initialize PM usage */
    hmq = WinCreateMsgQueue(hab, 0);         /* create message queue */

                                             /* register client window class */
    WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                             /* create standard window */
    hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
              szClientClass, " - Savemeta", 0L, NULL, ID_FRAMERC,
              &hwndClient);

                                             /* messages dispatch loop */
    while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
       WinDispatchMsg(hab, &qmsg);

    WinDestroyWindow(hwnd);                  /* destroy frame window */
    WinDestroyMsgQueue(hmq);                 /* destroy message queue */
    WinTerminate(hab);                       /* terminate PM usage */

    return 0;
    }

                                             /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
    {
    static HDC hdcWin;
    HDC hdcMetafile;
    static HPS hps;
    static SIZEL sizl = {0, 0};

    static POINTL ptl;
    int i;

    static POINTL ptlFenders[6] = {{145, 230}, {220, 160}, {220, 70},
                                   {450, 180}, {550, 210}, {545, 70}};

    static POINTL ptlWind[3] = {{265, 225}, {275, 225}, {310, 175}};

    static MATRIXLF matlfWheel2 = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                                   MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                                   330L, 0L, 1L};

    static MATRIXLF matlfTree = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                                 MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                                 500L};

    static POINTL branch[BRANCHES] = {{150,330}, {150,300},
                  {120,270}, {180,240}, {110,210}, {190,180},
                  {80,120}, {220,130}, {150,140}};

    static DEVOPENSTRUC dop = {NULL, "DISPLAY"};

    HMF hmf;
```

```
switch (msg)
   {
   case WM_PAINT:
      hps = WinBeginPaint(hwnd, hps, NULL);
      GpiErase(hps);

      GpiDrawChain(hps);            /* draw segment chain */

      WinEndPaint(hps);
      break;


   case WM_COMMAND:
      switch (COMMANDMSG(&msg) -> cmd)
         {
         case ID_SAVE:                /* save image to metafile */
                                      /* open metafile DC */
            hdcMetafile = DevOpenDC(hab, OD_METAFILE, "*", 2L,
                                      (PDEVOPENDATA)&dop, (HDC)NULL);

            GpiAssociate(hps, NULL);      /* disassociate PS from win */
            GpiAssociate(hps, hdcMetafile);   /* associate PS/meta DC */

            GpiDrawChain(hps);      /* draw segment chain */

            GpiAssociate(hps, NULL);      /* disassociate PS/metafile */
            GpiAssociate(hps, hdcWin);    /* associate with window DC */
            hmf = DevCloseDC(hdcMetafile);        /* close meta DC */

            GpiSaveMetaFile(hmf, "metafile.met");     /* save metafile */
            break;
         }
      break;

   case WM_CREATE:
      hdcWin = WinOpenWindowDC(hwnd);
      hps = GpiCreatePS(hab, hdcWin, &sizl, PU_LOENGLISH |
               GPIT_NORMAL | GPIA_ASSOC);

      GpiSetDrawingMode(hps, DM_RETAIN);  /* set drawing mode */

      GpiOpenSegment(hps, 0L);            /* open segment */
      GpiSetColor(hps, CLR_RED);          /* draw body rectangle */
      ptl.x = 125; ptl.y = 70;
      GpiMove(hps, &ptl);
      ptl.x = 505; ptl.y = 175;
      GpiBox(hps, DRO_OUTLINEFILL, &ptl, 0L, 0L);

      ptl.x = 300; ptl.y = 175;       /* draw windshield */
      GpiBeginArea(hps, BA_BOUNDARY);
      GpiMove(hps, &ptl);
      GpiPolyLine(hps, 3L, ptlWind);
      GpiEndArea(hps);

      ptl.x = 30; ptl.y = 70;         /* draw fenders */
      GpiMove(hps, &ptl);
      GpiSetColor(hps, CLR_DARKRED);
      GpiBeginArea(hps, BA_BOUNDARY);
      GpiPolySpline(hps, 6L, ptlFenders);
      GpiEndArea(hps);
```

```
    GpiCallSegmentMatrix(hps, ID_WHEEL_SEG, 0L, NULL,
       TRANSFORM_REPLACE);
    GpiCallSegmentMatrix(hps, ID_WHEEL_SEG, 7L, &matlfWheel2,
       TRANSFORM_REPLACE);

    GpiCloseSegment(hps);                   /* close segment */

                                            /* create wheel segment */
    GpiOpenSegment(hps, ID_WHEEL_SEG);              /* open segment */
    ptl.x = 150; ptl.y = 100;
    GpiMove(hps, &ptl);
    GpiSetColor(hps, CLR_BLACK);            /* outer tire */
    GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(55,0));
    GpiSetColor(hps, CLR_BACKGROUND);       /* spoke background */
    GpiFullArc(hps, DRO_FILL, MAKEFIXED(30,0));
    GpiSetColor(hps, CLR_RED);
    for(i = 0; i < 20; i++)                 /* inner tire and spokes */
       {
       GpiMove(hps, &ptl);
       GpiPartialArc(hps, &ptl, MAKEFIXED(30,0),
          MAKEFIXED(i * 18,0), MAKEFIXED(18,0));
       }
    GpiMove(hps, &ptl);                     /* hubcap */
    GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(15, 0));
    GpiCloseSegment(hps);                   /* close segment */

    GpiSetSegmentAttrs(hps, ID_WHEEL_SEG, ATTR_CHAINED, ATTR_OFF);

    GpiSetViewingTransformMatrix(hps, 7L, &matlfTree,
       TRANSFORM_REPLACE);

                                            /* create tree */
    GpiOpenSegment(hps, ID_TREE_SEG); /* open segment */

    ptl.x = 135; ptl.y = 0;                 /* draw trunk */
    GpiMove(hps, &ptl);
    ptl.x = 165; ptl.y = 100;
    GpiSetColor(hps, CLR_BROWN);
    GpiBox(hps, DRO_FILL, &ptl, 0L, 0L);

    for(i = 0; i < BRANCHES; i++)           /* draw branches */
       {
       GpiMove(hps, &branch[i]);
       GpiSetColor(hps, CLR_DARKGREEN);
       GpiFullArc(hps, DRO_FILL, MAKEFIXED(50,0));
       GpiSetColor(hps, CLR_GREEN);
       branch[i].x +=10; branch[i].y + = 10;
       GpiMove(hps, &branch[i]);
       GpiFullArc(hps, DRO_FILL, MAKEFIXED(30,0));
       }
    GpiCloseSegment(hps);          /* close segment */

    GpiSetDrawingMode(hps, DM_DRAW);        /* set drawing mode */
    break;

case WM_DESTROY:
   GpiDeleteSegments(hps, ID_WHEEL_SEG, ID_TREE_SEG);
   GpiAssociate(hps, NULL);
   GpiDestroyPS(hps);
   break;
```

```
      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                          /* NULL / FALSE */
   }
```

```
/* ---------- */
/* SAVEMETA.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_SAVE 100

#define ID_WHEEL_SEG 200L
#define ID_TREE_SEG 300L

#define BRANCHES 9
```

```
# ------------------
# SAVEMETA Make file
# ------------------

savemeta.obj: savemeta.c savemeta.h
   cl -c -G2s -W3 -Zp savemeta.c

savemeta.res: savemeta.rc savemeta.h
   rc -r savemeta.rc

savemeta.exe: savemeta.obj savemeta.def
   link /NOD savemeta,,NUL,os2 slibce,savemeta
   rc savemeta.res

savemeta.exe: savemeta.res
   rc savemeta.res
```

```
; ------------
; SAVEMETA.DEF
; ------------

NAME SAVEMETA WINDOWAPI

DESCRIPTION 'Save a metafile'

PROTMODE
STACKSIZE   4096
```

```
/* ----------- */
/* SAVEMETA.RC */
/* ----------- */

#include <os2.h>
#include "savemeta.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "~Save", ID_SAVE
   END
```

In this example a picture chain containing one car and one tree is created during WM_CREATE processing, just as it has been in many previous examples. Also as in previous examples, this chain is drawn in the client window whenever a WM_PAINT message is received.

Creating a Metafile

The metafile is created when the user selects "Save" from the menu. This generates a WM_COMMAND message with a command value of ID_SAVE.

To create a metafile, we use DevOpenDC to open a device context with a *type* of OD_METAFILE. In this respect a metafile is treated like a device, such as a printer. The metafile device context is then associated with a presentation space, using GpiAssociate. Since this same PS has already been associated with another device, the window, it must be disassociated from the window before it can be associated with the metafile. This is done using GpiAssociate with the second parameter set to NULL.

Once the PS is associated with the metafile, the picture is drawn into the metafile using the GpiDrawChain API. When the drawing is completed, the presentation space is disassociated from the metafile device context with GpiAssociate and reassociated with the window device context. The metafile device context is closed with DevCloseDC.

DevCloseDC also returns a metafile handle, *hmf.* This handle identifies the metafile and will be used in subsequent interactions with it.

Saving a Metafile to Disk

To write the metafile to a disk file, we execute GpiSaveMetaFile.

Save a Metafile to Disk

```
BOOL GpiSaveMetaFile(hmf, pszFilename)
HMF hmf          Handle to metafile
PSZ pszFilename  String containing path and filename

Returns: GPI_OK if successful, GPI_ERROR if error
```

The first argument to this function is the metafile handle obtained from DevCloseDC. The second argument is a zero-terminated string containing the filename specification of the file to which the metafile will be saved. By convention metafiles should have a .MET extension. Once the metafile is saved to disk, it is deleted from memory and is not available for use until it is reloaded.

Loading a Metafile from Disk

The LOADMETA example program loads the metafile from the disk and displays its contents on the screen. Here are the listings for LOADMETA.C, LOADMETA.H, LOADMETA, and LOADMETA.DEF.

```
/* -------------------------- */
/* LOADMETA.C Load a metafile */
/* -------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "loadmeta.h"

HAB hab;                               /* handle for anchor block */


USHORT cdecl main(void)
   {
   HMQ hmq;                            /* handle for message queue */
   QMSG qmsg;                          /* message queue element */
   HWND hwnd, hwndClient;              /* handles for windows */

                                       /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);          /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);    /* create message queue */
```

```
                                        /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                        /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Loadmeta", 0L, NULL, 0, &hwndClient);

                                        /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);              /* destroy frame window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */

   return 0;
   }

                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                     MPARAM mp2)
   {
   static HDC hdc;
   static HPS hps;
   static HMF hmf;

   static SIZEL sizl = {0, 0};
   static LONG alOptions[] = {0L, LT_ORIGINALVIEW, RS_DEFAULT,
                              LC_DEFAULT, RES_RESET};

   switch (msg)
      {
      case WM_PAINT:
         hps = WinBeginPaint(hwnd, hps, NULL);
         GpiErase(hps);
                                        /* play metafile */
         GpiPlayMetaFile(hps, hmf, 5L, alOptions, 0, 0L, NULL);

         WinEndPaint(hps);
         break;

      case WM_CREATE:
         hdc = WinOpenWindowDC(hwnd);
         hps = GpiCreatePS(hab, hdc, &sizl, PU_LOENGLISH |
                  GPIT_NORMAL | GPIA_ASSOC);
                                        /* load metafile */
         hmf = GpiLoadMetaFile(hab, "metafile.met");
         break;

      case WM_DESTROY:
         GpiDeleteMetaFile(hmf);
         GpiAssociate(hps, NULL);
         GpiDestroyPS(hps);
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }
```

```
   return NULL;                          /* NULL / FALSE */
   }
```

```
/* ---------- */
/* LOADMETA.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);
```

```
# ------------------
# Loadmeta Make file
# ------------------

loadmeta.obj: loadmeta.c loadmeta.h
   cl -c -G2s -W3 -Zp loadmeta.c

loadmeta.exe: loadmeta.obj loadmeta.def
   link /NOD loadmeta,,NUL,os2 slibce,loadmeta
```

```
; ------------
; LOADMETA.DEF
; ------------

NAME   LOADMETA WINDOWAPI

DESCRIPTION 'Load a metafile'

PROTMODE
STACKSIZE   4096
```

Loading the metafile from disk takes place during WM_CREATE processing. The API used to load the metafile is GpiLoadMetaFile.

Load Metafile from Disk

```
HMF GpiLoadMetaFile(hab, pszFilename)
HAB hab            Handle to anchor block
PSZ pszFilename    String containing path and filename

Returns: Handle to metafile if successful, GPI_ERROR if error
```

The first argument to this function is the anchor block handle, and the second is the file specifications of the metafile to be loaded. The function returns a handle to the metafile, which can be used in subsequent references to it.

Playing a Metafile

To display the contents of a metafile that has been loaded into memory, we execute GpiPlayMetaFile during WM_PAINT processing. This function has many options, which determine whether the attributes and transformations to be used will be those of the metafile itself, or those from the current presentation space.

Play Metafile

```
LONG GpiPlayMetaFile(hps, hmf, cOptions, alOptions, pcSegments,
                     cchDesc, pszDesc)
HPS hps             Handle to presentation space
HMF hmf             Handle to metafile
LONG cOptions       Number of elements in alOptions array
PLONG alOptions     Array of load options
PLONG pcSegments    Address for renumbered segment count
LONG cchDesc        Number of bytes in record
PSZ pszDesc         Descriptive record

Returns: GPI_OK or GPI_HITS if successful, GPI_ERROR if error
```

The first argument to this function is the presentation space into which the metafile will be played, and the second is the handle to the metafile. In our program this handle was returned from GpiLoadMetaFile.

The *cOptions* argument is the number of elements in the array *alOptions.* This array contains up to ten elements, each of which specifies one aspect of how the metafile will be played. The element is indicated by a predefined index value, or simply by an index number. The following condensed table shows the index values and the possible values for each.

Index/Identifier	**Meaning**
PMF_SEGBASE (0)	Reserved; must be 0L
PMF_LOADTYPE (1)	Transformation to use:
LT_DEFAULT	Same as LT_NONMODIFY
LT_NONMODIFY	Set by application
LT_ORIGINALVIEW	Set by metafile
PMF_RESOLVE (2)	Reserved; must be RS_DEFAULT
PMF_LCIDS (3)	Where bitmaps and fonts come from:
LC_DEFAULT	Same as LC_NOLOAD
LC_NOLOAD	From application

LC_LOADDISC	From metafile
PMF_RESET (4)	Reset PS before playing metafile?
RES_DEFAULT	Same as RES_NORESET
RES_NORESET	No
RES_RESET	Yes, to page units of metafile
PMF_SUPPRESS (5)	Suppress playing metafile after reset?
SUP_DEFAULT	Same as SUP_NOSUPPRESS
SUP_NOSUPPRESS	No
SUP_SUPPRESS	Yes
PMF_COLORTABLES (6)	Where logical color tables come from:
CTAB_DEFAULT	Same as CTAB_NOMODIFY
CTAB_NOMODIFY	Application
CTAB_REPLACE	Metafile
PMF_COLORREALIZABLE (7)	Metafile logical color table realizable?
CREA_DEFAULT	Same as CREA_NOREALIZE
CREA_REALIZE	Yes
CREA_NOREALIZE	No
PMF_PATHBASE (8)	Reserved; must be 0
PMF_DISSOLVEPATH (9)	Reserved; must be 0

In our example we override the default values for two identifiers. The first is the LT_ORIGINALVIEW identifier to the PMF_LOADTYPE element. This specifies that we use the viewing transformation defined in the metafile, rather than the viewing transform already set in the PS (which is the identity transform in this case). Second, the PMF_RESET identifier is set to RES_RESET. This specifies that the presentation space should be reset before playing the metafile.

The fifth argument to GpiPlayMetaFile, *pcSegments,* is not currently used and must be set to 0. The sixth argument, *cchDesc,* specifies the number of bytes in the buffer *pszDesc,* the seventh and last argument. This buffer holds a string describing the metafile. This string can be set when the metafile device is created with DevOpenDC, using the *pszToken* argument to this function. We don't use it in this example; if we did, the string might be "Car and tree."

You can exchange a metafile with another application by placing its handle on the clipboard. In the next chapter we'll see how the clipboard works.

IV

GLOBAL ORGANIZATION

15

THE CLIPBOARD

With this chapter we begin Part IV, the final section of the book. In this section we'll devote brief chapters to introducing several PM topics. We won't go into great detail on these subjects; we'll save that for more advanced PM books. However, we'll show some examples and try to give a feeling for what these topics are about.

This chapter introduces the clipboard. The clipboard is an area in memory that can be used for exchanging data between programs. The user can copy data from one application to the clipboard and then switch to another application and retrieve the data.

In one common situation, you might want to transfer a table of figures from a spreadsheet to a letter. From within the spreadsheet you would first copy the table to the clipboard. Then you would switch to your word processor, where you could "paste" or transfer the data from the clipboard into your letter.

The clipboard can handle graphics as well as text. For example, you could use it to transfer part of a picture from a paint program to a page layout program or to a word processor with graphics capabilities.

CLIPBOARD OVERVIEW

Before we present our example programs, we'll describe the clipboard in general terms, first from the user's viewpoint, and then from the programmer's.

Clipboard Usage

A program that utilizes the clipboard typically has a menu called "Edit" with the selections "Cut", "Copy", "Paste", and "Delete" (or "Clear"). To transfer data to the clipboard, you would first select the material to be transferred. In a text-based program this might involve dragging the mouse along the text to highlight it. In a graphics program you might use the mouse to position a rectangle around the appropriate part of a picture.

If you wanted to copy the data, leaving the original unchanged, you would select "Copy". The selected data would be written to the clipboard without affecting the original document. If you wanted to transfer the data to the clipboard and at the same time delete it from the original, you would select "Cut". You could also delete the selection without transferring it to the clipboard by selecting "Delete". ("Clear" is similar to "Delete", but does not compress the empty space left by the deleted material.)

The destination application might contain a similar menu. If you selected "Paste", the data would be transferred from the clipboard to your document, at the cursor position or another location you specified. This process is shown in Figure 15-1. (The terms *cut* and *paste* are carryovers from the time before word processors, when manuscripts were rearranged with scissors and glue.)

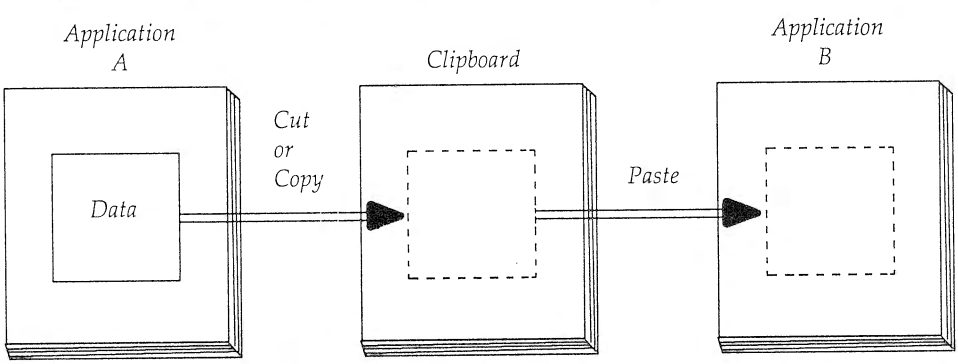


Figure 15-1. Typical clipboard usage

The clipboard need not be used solely for transfers between two separate programs. An application can cut or copy material to the clipboard, and then paste it back to itself, usually in a different location in the document. This is an easy way to add powerful editing capabilities to a program.

The clipboard is a remarkably useful and versatile feature of PM. Pictures, text, and other data now can be exchanged easily, using standard formats that all applications understand. Most applications can substantially increase their versatility by implementing clipboard operations.

Programming Considerations

Here are some points to keep in mind when writing programs that utilize the clipboard.

Illusory Data Storage

The user's conception is that text and graphics are stored in the clipboard. However, this isn't what really happens. An application can actually store only two kinds of objects in the clipboard: handles and selectors. Transferring handles and selectors allows different applications to access the same data. Handles provide access to metafiles and bitmaps, while selectors—segment selectors to shared memory—provide access to text and to custom data formats.

Only One Copy

Although different applications may access the same data using the clipboard, there is only one copy of the data. Handles or selectors to the data may be passed to different applications, but that does not mean that the data itself has been copied.

Serialized Access

Only one application at a time can be permitted to access clipboard data. Why is this restriction necessary? Imagine that the user tells application A to write something to the clipboard, and—before it is finished—switches to application B and attempts to read the same material. The results would be incomplete or garbled. To avoid this problem, an application must open the clipboard before reading or writing data to it. Once one application opens it, other applications can't open it until the first one has closed it. The application that opens the clipboard has exclusive access to clipboard data.

Once it closes the clipboard, an application can no longer access the data in it, even if it created the data in the first place. If an application wants to use data that has been placed in the clipboard after the clipboard is closed, it must create a private copy before closing it.

User Control

Clipboard operations must always be initiated by the user. An application should not access the clipboard except as the result of menu selections or other direct instructions by the user. Another PM feature, Dynamic Data Exchange (DDE), is used when data is exchanged under program control rather than user control. We'll mention DDE again at the end of this chapter.

Same Data, Different Formats

To make the clipboard as versatile as possible, an application placing data in it should do so using as many formats as possible. This will enable a wider variety of applications to access the data. For instance, a word processor might copy a text selection as ASCII, as a metafile, and as a proprietary format that included boldface and other information specific to that brand of word processor. Any word processor could read the ASCII file, a graphics program could read the metafile, and another instance of the original word processor (or the original instance) could read the proprietary format. The aim is to make data as available as possible, regardless of the type of application that may try to access it.

Several different formats can be placed in the clipboard simultaneously, but they should all represent the same data (the same text, in the case of the word processor).

Only One Clipboard

There is only one clipboard in the system. Since there is only one user, a single clipboard is all that is necessary because there is usually no inconvenience in completing one copy-and-paste operation before going on to the next.

Three Example Programs

Our clipboard demonstration consists of three different programs. COPYCLIP generates a picture of a car and tree, familiar from previous

examples. If the user selects "Copy" from the action bar, the program then copies this picture to the clipboard. It also copies a line of text, "A picture of a car and a tree", to the clipboard. This demonstrates that the clipboard can simultaneously hold data in several formats.

Once the picture is written to the clipboard, the PASTEMET program can retrieve the picture from the clipboard and display it. You can terminate COPYCLIP or leave it running if you wish; then start up PASTEMET. When you select the "Paste" menu item, PASTEMET will obtain the picture from the clipboard and display it.

The third program, PASTETXT, retrieves the text from the clipboard and displays it when the user selects "Paste" from its menu.

We'll first examine the steps necessary to copy a picture to the clipboard. That is, we'll look at the parts of COPYCLIP concerned with transferring graphics. Then we'll show how the PASTEMET program retrieves this picture from the clipboard. Next we'll examine the parts of COPYCLIP that copy text to the clipboard. And finally, we'll see how PASTETXT retrieves text from the clipboard.

COPYING A METAFILE TO THE CLIPBOARD

There are two standard ways to store graphics information in the clipboard: as metafiles, and as bitmaps. They use similar approaches, but the metafile is probably the most common, so we'll demonstrate it in the example. Here are the steps necessary to copy a metafile to the clipboard:

1. Open the clipboard with WinOpenClipbrd.
2. Empty the existing clipboard contents with WinEmptyClipbrd.
3. Place the metafile handle in the clipboard with WinSetClipbrdData.
4. Close the clipboard with WinCloseClipbrd.

The picture (actually, its handle) is now in the clipboard, where it can be accessed by other applications.

We saw in Chapter 13 how to create a metafile. In COPYCLIP the segments containing the picture are created during WM_CREATE processing, and the picture is drawn on the screen during WM_PAINT processing, using GpiDrawChain. The metafile is created from these segments when the user selects "Copy" from the action bar, thus generating a WM_COMMAND message. The program opens a metafile device context, associates it with the presentation space, draws the picture chain to it, and

closes the device context. This places the picture in the metafile. The handle returned from DevCloseDC, *hmf*, is used for subsequent references to the metafile. As we noted, it is not the metafile itself, but this metafile handle that will be placed in the clipboard.

Here are the listings for COPYCLIP.C, COPYCLIP.H, COPYCLIP, COPYCLIP.DEF, and COPYCLIP.RC.

```
/* -------------------------------- */
/* COPYCLIP.C Copy to the clipboard */
/* -------------------------------- */

#define INCL_GPI
#define INCL_WIN
#define INCL_DEV
#define INCL_DOS
#include <os2.h>

#include <string.h>

#include "copyclip.h"

HAB hab;                                /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd, hwndClient;               /* handles for windows */

                                        /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                      FCF_MINMAX | FCF_SHELLPOSITION | FCF_MENU |
                      FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);     /* create message queue */

                                        /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                        /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
            szClientClass, " - Copy to Clipboard", 0L, NULL, ID_FRAMERC,
            &hwndClient);

                                        /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);              /* destroy frame window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */

   return 0;
   }

                                        /* client window procedure */
```

```
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   static HDC hdcWin;
   HDC hdcMetafile;
   static HPS hps;
   static SIZEL sizl = {0, 0};
   static POINTL ptl;
   int i;

   static POINTL ptlFenders[6] = {{145, 230}, {220, 160}, {220, 70},
                                  {450, 180}, {550, 210}, {545, 70}};

   static POINTL ptlWind[3] = {{265, 225}, {275, 225}, {310, 175}};

   static MATRIXLF matlfWheel2 = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                                  MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                                  330L, 0L, 1L};

   static MATRIXLF matlfTree = {MAKEFIXED(1, 0), MAKEFIXED(0, 0), 0L,
                                MAKEFIXED(0, 0), MAKEFIXED(1, 0), 0L,
                                500L};

   static POINTL branch[BRANCHES] = {{150, 330}, {150, 300},
                    {120, 270}, {180, 240}, {110, 210}, {190, 180},
                    {80, 120}, {220, 130}, {150, 140}};

   static DEVOPENSTRUC dop = {NULL, "DISPLAY"};

   HMF hmf;
   SEL sel;
   static CHAR szString[] = "A picture of a car and a tree";
   PSZ pszTo;
   NPSZ npszFrom;

   switch (msg)
      {
      case WM_PAINT:
         hps = WinBeginPaint(hwnd, hps, NULL);
         GpiErase(hps);

         GpiDrawChain(hps);            /* draw segment chain */

         WinEndPaint(hps);
         break;

      case WM_COMMAND:
         switch (COMMANDMSG(&msg) -> cmd)
            {
            case ID_COPY:              /* copy to clipboard */

                                                /* open metafile DC */
               hdcMetafile = DevOpenDC(hab, OD_METAFILE, "*", 2L,
                                       (PDEVOPENDATA)&dop, (HDC)NULL);
               GpiAssociate(hps, NULL);     /* disassociate PS from win */
               GpiAssociate(hps, hdcMetafile);   /* associate PS, DC */
               GpiDrawChain(hps);           /* draw segment chain */
               GpiAssociate(hps, NULL);     /* disassociate PS, DC */
               GpiAssociate(hps, hdcWin);   /* associate with window DC */
               hmf = DevCloseDC(hdcMetafile);      /* close printer DC */
```

```
                                        /* alloc giveable seg */
            DosAllocSeg(strlen(szString) + 1, &sel, SEG_GIVEABLE);
            pszTo = MAKEP(sel, 0);
            npszFrom = &szString[0];
            while (*pszTo++ = *npszFrom++);    /* copy string */

            WinOpenClipbrd(hab);        /* open clipboard */
            WinEmptyClipbrd(hab);       /* empty clipboard */
                                        /* put metafile in clipboard */
            WinSetClipbrdData(hab, (ULONG)hmf, CF_METAFILE, CFI_HANDLE);
                                        /* put text in clipboard */
            WinSetClipbrdData(hab, (ULONG)sel, CF_TEXT, CFI_SELECTOR);
            WinCloseClipbrd(hab);       /* close clipboard */
            break;
         }
      break;

   case WM_CREATE:
      hdcWin = WinOpenWindowDC(hwnd);   /* get window DC */
                                        /* and a PS */
      hps = GpiCreatePS(hab, hdcWin, &sizl, PU_LOENGLISH |
               GPIT_NORMAL | GPIA_ASSOC);

      GpiSetDrawingMode(hps, DM_RETAIN);
      GpiOpenSegment(hps, 0L);          /* open segment */
      GpiSetColor(hps, CLR_RED);        /* draw body rectangle */
      ptl.x = 125; ptl.y = 70;
      GpiMove(hps, &ptl);
      ptl.x = 505; ptl.y = 175;
      GpiBox(hps, DRO_OUTLINEFILL, &ptl, 0L, 0L);

      ptl.x = 300; ptl.y = 175;         /* draw windshield */
      GpiBeginArea(hps, BA_BOUNDARY);
      GpiMove(hps, &ptl);
      GpiPolyLine(hps, 3L, ptlWind);
      GpiEndArea(hps);

      ptl.x = 30; ptl.y = 70;           /* draw fenders */
      GpiMove(hps, &ptl);
      GpiSetColor(hps, CLR_DARKRED);
      GpiBeginArea(hps, BA_BOUNDARY);
      GpiPolySpline(hps, 6L, ptlFenders);
      GpiEndArea(hps);

      GpiCallSegmentMatrix(hps, ID_WHEEL_SEG, 0L, NULL,
         TRANSFORM_REPLACE);
      GpiCallSegmentMatrix(hps, ID_WHEEL_SEG, 7L, &matlfWheel2,
         TRANSFORM_REPLACE);

      GpiCloseSegment(hps);             /* close segment */

                                        /* create wheel segment */
      GpiOpenSegment(hps, ID_WHEEL_SEG);            /* open segment */
      ptl.x = 150; ptl.y = 100;
      GpiMove(hps, &ptl);
      GpiSetColor(hps, CLR_BLACK);      /* outer tire */
      GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(55, 0));
      GpiSetColor(hps, CLR_BACKGROUND);  /* spoke background */
      GpiFullArc(hps, DRO_FILL, MAKEFIXED(30, 0));
      GpiSetColor(hps, CLR_RED);
```

```
    for(i = 0; i < 20; i++)            /* inner tire and spokes */
       {
       GpiMove(hps, &ptl);
       GpiPartialArc(hps, &ptl, MAKEFIXED(30, 0),
          MAKEFIXED(i * 18, 0), MAKEFIXED(18, 0));
       }
    GpiMove(hps, &ptl);                /* hubcap */
    GpiFullArc(hps, DRO_OUTLINEFILL, MAKEFIXED(15, 0));
    GpiCloseSegment(hps);              /* close segment */

    GpiSetSegmentAttrs(hps, ID_WHEEL_SEG, ATTR_CHAINED, ATTR_OFF);

    GpiSetViewingTransformMatrix(hps, 7L, &matlfTree,
       TRANSFORM_REPLACE);

                                       /* create tree */
    GpiOpenSegment(hps, ID_TREE_SEG);  /* open segment */

    ptl.x = 135; ptl.y = 0;            /* draw trunk */
    GpiMove(hps, &ptl);
    ptl.x = 165; ptl.y = 100;
    GpiSetColor(hps, CLR_BROWN);
    GpiBox(hps, DRO_FILL, &ptl, 0L, 0L);

    for(i = 0; i < BRANCHES; i++)      /* draw branches */
       {
       GpiMove(hps, &branch[i]);
       GpiSetColor(hps, CLR_DARKGREEN);
       GpiFullArc(hps, DRO_FILL, MAKEFIXED(50, 0));
       GpiSetColor(hps, CLR_GREEN);
       branch[i].x += 10; branch[i].y += 10;
       GpiMove(hps, &branch[i]);
       GpiFullArc(hps, DRO_FILL, MAKEFIXED(30, 0));
       }
    GpiCloseSegment(hps);              /* close segment */

    GpiSetDrawingMode(hps, DM_DRAW);   /* set drawing mode */
    break;

 case WM_DESTROY:
    GpiDeleteSegments(hps, ID_WHEEL_SEG, ID_TREE_SEG);
    GpiAssociate(hps, NULL);
    GpiDestroyPS(hps);
    break;

 default:
    return(WinDefWindowProc(hwnd, msg, mp1, mp2));
    break;
 }

return NULL;                           /* NULL / FALSE */
}
```

```
/* ---------- */
/* COPYCLIP.H */
/* ---------- */
```

```
USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_COPY 100

#define ID_WHEEL_SEG 200L
#define ID_TREE_SEG 300L

#define BRANCHES 9
```

```
# ------------------
# COPYCLIP Make file
# ------------------

copyclip.obj: copyclip.c copyclip.h
   cl -c -G2s -W3 -Zp copyclip.c

copyclip.res: copyclip.rc copyclip.h
   rc -r copyclip.rc

copyclip.exe: copyclip.obj copyclip.def
   link /NOD copyclip,,NUL,os2 slibce,copyclip
   rc copyclip.res

copyclip.exe: copyclip.res
   rc copyclip.res
```

```
; ------------
; COPYCLIP.DEF
; ------------

NAME        COPYCLIP WINDOWAPI

DESCRIPTION 'Copy to the clipboard'

PROTMODE
STACKSIZE   4096
```

```
/* ----------- */
/* COPYCLIP.RC */
/* ----------- */

#include <os2.h>
#include "copyclip.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "~Copy", ID_COPY
   END
```

For the time being we'll ignore the code that places text in a shared memory segment and concentrate on copying the metafile to the clipboard. Placing data in the clipboard requires several Win functions. Let's look at them.

Opening the Clipboard

Before an application can use the clipboard, it must open it. This is accomplished by the WinOpenClipbrd function. This function is somewhat unusual. To understand why, remember that only one application can access clipboard data at once. If this weren't true, the clipboard contents could be altered simultaneously by several different programs, leading to misunderstandings. To avoid such problems, WinOpenClipbrd will not return if the clipboard is already open. It waits until the clipboard has been closed by any other application using it.

This may seem like a reasonable way to handle conflict over the clipboard, but now imagine that the application that is using the clipboard is keeping it open for a long time. Does this mean our application will be immobilized until WinOpenClipbrd returns? If so, problems could arise if any messages are posted to our application, since we must be able to respond to messages in a fraction of a second. Fortunately, WinOpenClipbrd permits messages to arrive at our application even though it has not yet returned. That is, your program can be doing two things at once: waiting for WinOpenClipbrd, and processing a message. Of course, as we've noted in earlier chapters, this means you must write your program to be reentrant, but you should do this anyway.

Open the Clipboard

```
BOOL WinOpenClipbrd(hab)
HAB hab   Anchor block handle

Returns: TRUE if successful, FALSE if error
```

Emptying the Clipboard

Before an application writes to the clipboard, it should empty the clipboard's old contents. Why is this necessary? As we mentioned, the same

data may be stored in different formats (as a metafile, as text, and so on). Writing new data to the clipboard will replace any existing data in the same format, but data in other formats is not removed. All the formats stored in the clipboard at a particular time should represent the same data, so it's safest to erase everything before writing. WinEmptyClipbrd clears all data, regardless of format.

Empty the Clipboard

```
BOOL WinEmptyClipbrd(hab)
HAB hab    Anchor block handle

Returns: TRUE if successful, FALSE if error
```

Placing Data in the Clipboard

The WinSetClipbrdData function is used to place data in the clipboard.

Write Data to the Clipboard

```
BOOL WinSetClipbrdData(hab, ulData, fmt, fsFmtInfo)
HAB hab             Anchor block handle
ULONG ulData        Data object (handle or selector)
USHORT fmt          Data format (CF_BITMAP, etc.)
USHORT fsFmtInfo    Data type (CFI_HANDLE, CFI_SELECTOR, etc.)

Returns: TRUE if successful, FALSE if error
```

For a metafile, the *ulData* argument to this function is set to the metafile handle. For text, it's set to a segment selector value, as we'll discuss later. The *fmt* argument tells what kind of data is being transferred. It can have the following values:

Identifier	Data Format
CF_BITMAP	Bitmap
CF_DSPBITMAP	Bitmap, display format
CF_METAFILE	Metafile
CF_DSPMETAFILE	Metafile, display format
CF_TEXT	Text
CF_DSPTEXT	Text, display format

We use CF_METAFILE in the call to WinSetClipbrdData that transfers the metafile. (CF_TEXT is used in the call that transfers text.)

The *fsFmtInfo* argument specifies whether the data consists of a handle, specified by CFI_HANDLE, or a segment selector, specified by CFI_SELECTOR. Handles are used for metafiles and selectors for text. Two other constants can be ORed onto this argument: CFI_OWNERDISPLAY, and CFI_OWNERFREE. These are related to the concepts of clipboard viewers and clipboard owners, which we'll discuss later.

Closing the Clipboard

When data has been transferred to the clipboard, the clipboard must be closed before any other application can access it. This is accomplished with WinCloseClipbrd.

Close the Clipboard

```
BOOL WinCloseClipbrd(hab)
HAB hab     Anchor block handle

Returns: TRUE if successful, FALSE if error
```

That's all there is to it. The metafile handle is now in the clipboard, where it can be retrieved by other applications. The originating application can go about its business.

Remember that once a handle or segment selector has been passed to the clipboard, the data it represents is no longer the property of the application that created it. It belongs to the clipboard. The application

should not try to alter the data in any way. If the application needs to use the data at a later time, it should make a copy of the data and send the handle or selector of the copy to the clipboard.

While our example program, in the interest of simplicity, copies an entire metafile to the clipboard, an application might well want to copy only part of the picture. The user could position a rectangle around the part of the picture to be copied, and the program could save this part of the picture as a bitmap.

PASTING A METAFILE FROM THE CLIPBOARD

Our PASTEMET example pastes (retrieves) the metafile from the clipboard and displays it on the screen. Here are the listings for PASTEMET.C, PASTEMET.H, PASTEMET, PASTEMET.DEF, and PASTEMET.RC:

```
/* -------------------------------------------- */
/* PASTEMET.C Paste a metafile from the clipboard */
/* -------------------------------------------- */

#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "pastemet.h"

HAB hab;                              /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                           /* handle for message queue */
   QMSG qmsg;                         /* message queue element */
   HWND hwnd, hwndClient;             /* handles for windows */

                                      /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_MENU |
                         FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);         /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);   /* create message queue */

                                      /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                      /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
            szClientClass, " - Paste Metafile", 0L, NULL, ID_FRAMERC,
            &hwndClient);

                                      /* messages dispatch loop */
```

```
    while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
       WinDispatchMsg(hab, &qmsg);

    WinDestroyWindow(hwnd);              /* destroy frame window */
    WinDestroyMsgQueue(hmq);             /* destroy message queue */
    WinTerminate(hab);                   /* terminate PM usage */

    return 0;
    }

                                         /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
    {
    static HDC hdc;
    static HPS hps;
    static HMF hmf = NULL;
    HMF hmfClipbrd;

    static SIZEL sizl = {0, 0};

    static LONG alOptions[] = {0L, LT_ORIGINALVIEW, RS_DEFAULT,
                               LC_DEFAULT, RES_RESET};

    switch (msg)
       {
       case WM_PAINT:
          hps = WinBeginPaint(hwnd, hps, NULL);
          GpiErase(hps);
                                         /* play metafile */
          if (hmf != NULL)
             GpiPlayMetaFile(hps, hmf, 5L, alOptions, NULL, 0L, NULL);

          WinEndPaint(hps);
          break;

       case WM_COMMAND:
          switch (COMMANDMSG(&msg) -> cmd)
             {
             case ID_PASTE:              /* paste */
                WinOpenClipbrd(hab);     /* open clipboard */
                                         /* if there is a metafile */
                if (hmfClipbrd = WinQueryClipbrdData(hab, CF_METAFILE))
                   hmf = GpiCopyMetaFile(hmfClipbrd);   /* make a copy */

                WinCloseClipbrd(hab);    /* close metafile */
                                         /* go repaint window */
                WinInvalidateRect(hwnd, NULL, FALSE);
                break;
             }
          break;

       case WM_CREATE:
          hdc = WinOpenWindowDC(hwnd);
          hps = GpiCreatePS(hab, hdc, &sizl, PU_LOENGLISH |
                   GPIT_NORMAL | GPIA_ASSOC);
          break;

       case WM_DESTROY:
          GpiDeleteMetaFile(hmf);        /* destroy metafile */
          break;
```

```
      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                        /* NULL / FALSE */
   }
```

```
/* ---------- */
/* PASTEMET.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_PASTE 100
```

```
# ------------------
# PASTEMET Make file
# ------------------

pastemet.obj: pastemet.c pastemet.h
   cl -c -G2s -W3 -Zp pastemet.c

pastemet.res: pastemet.rc pastemet.h
   rc -r pastemet.rc

pastemet.exe: pastemet.obj pastemet.def
   link /NOD pastemet,,NUL,os2 slibce,pastemet
   rc pastemet.res

pastemet.exe: pastemet.res
   rc pastemet.res
```

```
; ------------
; PASTEMET.DEF
; ------------

NAME         PASTEMET WINDOWAPI

DESCRIPTION 'Paste a metafile from the clipboard'

PROTMODE
STACKSIZE   4096
```

```
/* ----------- */
/* PASTEMET.RC */
/* ----------- */
```

```
#include <os2.h>
#include "pastemet.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "~Paste", ID_PASTE
   END
```

As you can see, there's not much to this program. A window device context is opened during WM_CREATE processing so the program will have a place to put the metafile once it has pasted it.

Reading Clipboard Data

Pasting the metafile takes place when "Paste" is selected from the action bar. This results in a WM_COMMAND message. During processing of this message, WinOpenClipbrd opens the clipboard. Once this function has successfully returned, we need to retrieve the metafile handle. The WinQueryClipbrdData function does this.

Obtain Handle to Clipboard Data

```
ULONG WinQueryClipbrdData(hab, fmt)
HAB hab          Anchor block handle
USHORT fmt       Data format (CF_BITMAP, etc.)

Returns: Handle to data or NULL, indicating no data of the specified
         format (or error)
```

The *fmt* argument to this function can be given the same values as the *fmt* argument to WinSetClipbrdData, described earlier. In our example we use CF_METAFILE, since we want to retrieve a metafile. If data of the specified format exists in the clipboard, WinQueryClipbrdData returns a handle (or a selector, for text) to the data. If no data is stored in the specified format, WinQueryClipbrdData returns NULL. Our application checks the return value. If it's not NULL, it creates a private copy of the metafile using GpiCopyMetaFile and invalidates the window, so as to generate a WM_PAINT message. When the WM_PAINT message is received, GpiPlayMetaFile is used to display the picture.

If our application suspected there was data stored in several different formats, it could try WinQueryClipbrdData using different values for *fmt*. It would start with the preferred format, and if NULL was returned, it could then try again with the next most desirable format. For instance, a word processor might look first for its own private format, then for an ASCII text format, and finally for a bitmap or metafile. (Another function, WinQueryClipbrdFmtInfo, can be used to determine if data of a particular data format exists in the clipboard, without reading it.)

Retrieving data from the clipboard does not destroy the data. It remains in the clipboard, where it can be accessed by other applications. The data will be destroyed only when an application executes WinEmptyClipbrd or writes data in the same format to the clipboard.

Copying the Metafile

The metafile handle obtained from the clipboard is valid only as long as the clipboard remains open. The metafile belongs to the clipboard, not to our application. But in our example we want to be able to display the picture even after we close the clipboard, relinquishing control of the clipboard to other applications. To solve this problem, we create a copy of the clipboard's metafile using the GpiCopyMetaFile function.

Copy a Metafile

```
HMF GpiCopyMetaFile(hmfSrc)
HMF hmfSrc       Handle to original metafile

Returns: Handle to new metafile, or GPI_ERROR if error
```

This function returns a handle to the new metafile. Using this handle, we can play or otherwise operate on the new metafile no matter what the clipboard does with the old one. An application pasting data from the clipboard should make use of the data (display it or whatever), or else copy it, before closing the clipboard.

In the example, the new metafile is displayed during WM_PAINT processing, using the GpiPlayMetaFile function.

COPYING TEXT TO THE CLIPBOARD

Let's return to COPYCLIP. We've already seen how it copies pictures to the clipboard; this time we'll see how it copies text.

Allocating a Shared Segment

When the user selects "Copy" from the action bar, COPYCLIP, as we saw earlier, first obtains a metafile device context for the picture. Following this, it obtains a sharable memory segment for the text. This is handled with the DosAllocSeg kernel function.

Allocate Memory Segment

```
USHORT DosAllocSeg(usSize, psel, fsAttr)
USHORT usSize     Size of segment, in bytes (1 to 65,536)
PSEL psel         Address for segment selector
USHORT fsAttr     Segment attributes

Returns: 0 if successful, or error value
```

The *fsAttr* argument can be given the following values:

Identifier	Segment Characteristics
SEG_DISCARDABLE	Discardable, nonsharable
SEG_GETTABLE	Can be accessed with DosGetSeg
SEG_GIVEABLE	Can be given away with DosGiveSeg
SEG_NONSHARED	Nonsharable, nondiscardable
SEG_SIZEABLE	Can be made smaller with DosReallocSeg

We use the SEG_GIVEABLE identifier with this function so that the memory segment obtained can be shared with other applications.

Why do we need shared memory for text, when we didn't for a metafile? When a metafile handle is placed in the clipboard, the metafile is stored in a system area and can be accessed by any application that has been given the metafile's handle. Text, on the other hand, is normally

stored in a local data segment that can be referenced only by the application that created it. To transfer text between applications, we must copy it to a shared memory segment, where it can be accessed by applications other than its originator.

In our example, the text is the string *szString,* which has the constant value "A picture of a car and a tree". The shared segment obtained should be as long as this string.

Transferring Text to the Shared Segment

DosAllocSeg returns the segment selector in the argument *sel.* Now we want to copy the string from our program into the shared segment. First, we need a far pointer to the segment, so we use the macro MAKEP to create the pointer *pszTo,* using the selector and an offset value of 0. The *npszFrom* near pointer is set to the beginning of *szString,* and a *while* loop transfers the string, character by character, from *szString* to the shared segment.

Notice that we allocate the shared segment and transfer the text to it before opening the clipboard. In general we want to minimize the time the clipboard is open, so other applications will not unnecessarily be denied access to it. To do this, we perform any preliminary activities before accessing it.

Placing the Selector in the Clipboard

As we saw earlier, the clipboard is opened with WinOpenClipbrd, and its previous contents are cleaned out with WinEmptyClipbrd. WinSetClipbrdData is used to write the segment selector for the text into the clipboard, so its *fmt* argument is set to CF_TEXT and its *fsFmtInfo* argument is set to CFI_SELECTOR. Now the text (or more accurately, a selector to it) is in the clipboard, where it can be accessed by other applications.

PASTING TEXT FROM THE CLIPBOARD

The PASTETXT program demonstrates how to retrieve text from the clipboard. Here are the listings for PASTETXT.C, PASTETXT.H, PASTETXT, PASTETXT.DEF, and PASTETXT.RC:

```
/* ------------------------------------ */
/* PASTETXT.C Paste text from a metafile */
/* ------------------------------------ */
```

```
#define INCL_GPI
#define INCL_WIN
#include <os2.h>

#include "pastetxt.h"

HAB hab;                              /* handle for anchor block */

USHORT cdecl main(void)
   {
   HMQ hmq;                           /* handle for message queue */
   QMSG qmsg;                         /* message queue element */
   HWND hwnd, hwndClient;             /* handles for windows */

                                      /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_MENU |
                         FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);         /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);   /* create message queue */

                                      /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                      /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
            szClientClass, " - Paste Text", 0L, NULL, ID_FRAMERC,
            &hwndClient);

                                      /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);            /* destroy frame window */
   WinDestroyMsgQueue(hmq);           /* destroy message queue */
   WinTerminate(hab);                 /* terminate PM usage */

   return 0;
   }

                                      /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                     MPARAM mp2)
   {
   HPS hps;
   SEL sel;
   static LONG cszString = 0;
   PSZ pszTo, pszStringClipbrd;
   static CHAR szString[100];
   static POINTL ptl = {5, 5};

   switch (msg)
      {
      case WM_PAINT:
         hps = WinBeginPaint(hwnd, NULL, NULL);
         GpiErase(hps);
                                      /* draw text */
```

```
        GpiCharStringAt(hps, &ptl, cszString, &szString[0]);
        WinEndPaint(hps);
        break;

     case WM_COMMAND:
        switch (COMMANDMSG(&msg) -> cmd)
           {
           case ID_PASTE:              /* paste */
              WinOpenClipbrd(hab);     /* open clipboard */
                                       /* if there is text */
              if (sel = WinQueryClipbrdData(hab, CF_TEXT))
                 {                     /* copy it */
                 pszStringClipbrd = MAKEP(sel, 0);
                 pszTo = &szString[0];
                 while (*pszTo++ = *pszStringClipbrd++);
                 cszString = pszTo - &szString[0];
                 }
              WinCloseClipbrd(hab);    /* close clipboard */
                                       /* go redraw window */
              WinInvalidateRect(hwnd, NULL, FALSE);
              break;
           }
        break;

     default:
        return(WinDefWindowProc(hwnd, msg, mp1, mp2));
        break;
     }

  return NULL;                         /* NULL / FALSE */
  }
```

```
/* ---------- */
/* PASTETXT.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_PASTE 100
```

```
# ------------------
# PASTETXT Make file
# ------------------

pastetxt.obj: pastetxt.c pastetxt.h
   cl -c -G2s -W3 -Zp pastetxt.c

pastetxt.res: pastetxt.rc pastetxt.h
   rc -r pastetxt.rc
```

```
pastetxt.exe: pastetxt.obj pastetxt.def
   link /NOD pastetxt,,NUL,os2 slibce,pastetxt
   rc pastetxt.res

pastetxt.exe: pastetxt.res
   rc pastetxt.res
```

```
; ------------
; PASTETXT.DEF
; ------------

NAME        PASTETXT WINDOWAPI

DESCRIPTION 'Paste text from a metafile'

PROTMODE
STACKSIZE   4096
```

```
/* ----------- */
/* PASTETXT.RC */
/* ----------- */

#include <os2.h>
#include "pastetxt.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "~Paste", ID_PASTE
   END
```

Upon receipt of a WM_COMMAND message, caused by the user selecting "Paste" from the action bar, this program opens the clipboard with WinOpenClipbrd just as PASTEMET did. It then checks for the desired kind of data with WinQueryClipbrdData.

Assuming that text data is available, it must be copied from the shared memory segment to PASTETXT's private memory space, so that it can be used after the clipboard is closed. First, the selector returned from WinQueryClipbrd is made into a pointer with MAKEP. Then the string is copied into the application's space using a *while* loop, in the same way we copied the string to the shared segment in COPYCLIP. Finally, the length of the string is obtained by subtracting the address of the beginning of the string from the pointer to the end of the string.

With the string safely stored, we can now display it whenever we receive a WM_PAINT message. This is done with the GpiCharStringAt function, which draws the string at the point (5, 5), using the pointer and string length previously calculated.

OTHER CLIPBOARD OPERATIONS

Our examples demonstrate a straightforward approach to using the clipboard to transfer metafiles and text, but various refinements are possible. Let's briefly discuss some of these.

Bitmaps

A bitmap handle can be passed in exactly the same way that we passed the metafile handle in our examples. This makes it possible to transfer bitmaps conveniently from one application to another. While metafiles are commonly transferred between draw-type programs, bitmaps are the medium of exchange for paint-type programs.

Proprietary Formats

The clipboard deals with text, metafiles, and bitmaps in a standardized way. What happens when an application wants to use another kind of data? Each private data format must be given an ID number, so it can be recognized by the various applications that might want to use it. The way to generate such an ID is to use the system's *atom table.* An application creates a name like "BrandXTextFormat" and sends it to the atom table. The atom table creates a unique 16-bit integer called an *atom* based on this name. Other applications, if they know the name, can find out the ID from the atom table. The ID is then used by any application that wants to access the clipboard with this private format.

Proprietary data can be passed through a shared memory segment, as text is.

The Clipboard Viewer

A *clipboard viewer* is a window that continuously displays the contents of the clipboard. A clipboard viewer might be used in a stand-alone utility program. You could use such a utility to verify what you copied to the clipboard, or to remind you what was already there. Some applications, such as word processors, might want to keep a clipboard viewer window open on the clipboard at all times, for the convenience of the user. The clipboard viewer monitors the clipboard's contents without changing them. Only one clipboard viewer can exist in the system at any one time.

The clipboard viewer should be able to display the three standard formats: CF_BITMAP, CF_METAFILE, CF_TEXT. But imagine that a program stores data in a private format that the clipboard viewer does not recognize. How can the clipboard viewer display such data? The solution is for the program to store, in addition to the private format, an approximation of the data in a recognizable format. This approximation data is for display purposes only; it is not intended for data exchange. Therefore it should not have one of the three standard formats. Instead it uses one of three special display-only formats: CF_DSPBITMAP, CF_DSPMETAFILE, or CF_DSPTEXT.

To act as a clipboard viewer, a window procedure needs to be informed of changes to the clipboard. The clipboard viewer window procedure receives WM_DRAWCLIPBOARD messages whenever the clipboard contents change. It should then display the new contents of the clipboard. It will also want to redisplay the clipboard contents whenever it receives a WM_PAINT message.

Data Rendering

It may happen that an application finds it time-consuming to create data in many different formats on the off chance that another application will want to paste data in one of these formats. For instance, an application might not want to take the time to routinely translate a metafile into a bitmap. On the other hand, it might be willing to do the translation if it knew that another application was definitely interested in the bitmap format.

What's needed here is a way for the first application to leave a message in the clipboard that says in effect, "There's no data of this kind ready yet, but if you want it, I'll generate it." This delayed generation of a particular data format is called *delayed rendering*.

To indicate that it's prepared to do delayed rendering, an application calls WinSetClipbrdData, using a particular data format, with the *ulData* argument set to NULL. To find out if a second application is interested in the data, the first application must be able to receive WM_RENDERFMT messages. To do this, it must register itself as the *clipboard owner*.

When it receives WM_RENDERFMT, the first application renders (creates) the data in the requested format and sends its handle to the clipboard, using WinSetClipbrdData. The data can now be accessed by the second application, although it had to wait for the first application to generate it.

An application registers itself as the clipboard owner with the WinSetClipbrdOwner function. This enables it to receive messages concerning the clipboard's operation.

DYNAMIC DATA EXCHANGE

Dynamic data exchange (DDE) provides another way to transfer data between programs. However, DDE is used in a different type of situation from the clipboard. While the clipboard is controlled by the user, DDE is controlled by the application.

A typical use for DDE is automatically updating a document when information changes. For example, you might have a spreadsheet that contains figures for sales in different regions of the country. You might also have a financial document that included these figures. DDE provides a way to link the figures in the spreadsheet with those in the document, so that if the spreadsheet figures change, the corresponding figures in the document will automatically be changed, too. Or, you might have a graph of the same sales data; altering the spreadsheet would automatically generate a new graph using the revised data.

DDE is particularly valuable for integrated software packages that combine several different applications, such as word processing, spreadsheets, data bases, and graphics. In such applications DDE can act as the glue that allows the different applications to communicate.

DDE uses a protocol—a series of messages between the server program that provides the data and the client program that needs it. This protocol can be quite complicated. For instance, before data can be exchanged, the server must establish that the client is an appropriate application, and that it can handle the kind of data to be sent.

The actual data to be exchanged in a DDE transfer is stored in shared memory, much as text and proprietary data is in clipboard transfers.

16

MULTITASKING AND OBJECTS

In previous chapters we mentioned situations in which an application—and the entire system—can become temporarily "locked up" because it is unable to respond to messages. This happens when a programming task, such as drawing a complex graphics shape or sorting a file, takes a long time. We mentioned that one solution to this problem is multitasking. In this chapter we'll show how to implement multitasking using multiple threads, so that an application can respond to messages in a timely fashion, and at the same time continue with other processing.

We'll show three approaches to using multiple threads. The first is very simple: Two threads run simultaneously, but do not require any synchronization. In the second, two threads are synchronized using semaphores. In the third, one of the two threads creates something called an *object window:* a special kind of window used to perform non-visual tasks.

Although we discuss multitasking here in relation to responding rapidly to messages, this is only one of many reasons to use multitasking.

You won't need previous knowledge of kernel programming to understand this chapter, but if you want to explore multitasking in general, you should read a kernel programming book, such as one of those mentioned in Chapter 1.

THE PROBLEM

As we mentioned in Chapter 5, PM must serialize all user-input-related messages. Each such message must be processed before the next one can be acted on. This means that each application must process such messages as quickly as possible: usually within one-tenth of a second.

Now let's imagine that an application must carry out a task that takes longer than this. The task might be recalculating a spreadsheet, switching to a different page in a word processor document, reading or writing a disk file, drawing a picture, sorting a database file, calculating pi to 100,000 places, or any other time-consuming task. While the procedure is carrying out this task, it can't respond to messages.

The application should tell the system it's ready for messages by executing WinGetMsg in the message loop. But it can't execute this function until WinDispatchMsg returns, and this function can't return until the window procedure finishes what it's doing. No user-related messages in the system will be processed. The result for the user is that the computer appears to lock up. You can't select anything from your menus, you can't terminate your application, and you can't even hot-key to another session. This goes on until the application finishes its task. If the task is long, the user will not be happy. (Actually PM will notice if the application does not respond for a long time after the user pressed the hot-key, and will allow the user to terminate the non-responsive application, thus providing an escape hatch.)

There are several ways to execute a lengthy task while at the same time allowing your application to process messages. One possibility is to divide your task into many parts. After each part is completed, your procedure returns to the message loop, so that messages can be processed normally. The question then is how to initiate the next part of your processing. You might use the timer for this, doing a little processing every time you get a timer message. However, such solutions carry high overhead and can become excessively complex.

A more efficient—and elegant—approach is to process the task *at the same time* that you process messages. OS/2 makes this possible by providing multiple *threads.*

What is a thread? C programmers can think of a thread as a function that executes at the same time as other functions. More formally it is an entity that receives slices of CPU time: It's called a *thread of execution.* The concept of threads will be made clearer in our first example.

THE MULTIPLE THREADS SOLUTION

Our example program plays several bars of music as soon as it is executed. If we had used the usual program architecture, where the tune was played during processing of the WM_CREATE message (or any other message), the application and the system would have been paralyzed until the tune finished. Here, however, we use two threads: one to process messages, and one to play the tune. You can operate the application's window normally, or interact with other applications, while the tune is being played.

Here are the listings for TUNE.C, TUNE.H, TUNE, and TUNE.DEF.

```
/* ------------------ */
/* TUNE.C Play a tune */
/* ------------------ */

#define INCL_WIN
#define INCL_DOS

#include <os2.h>

#include "tune.h"

HAB hab;                                /* handle for anchor block */


USHORT cdecl main(void)
   {
   HMQ hmq;                             /* handle for message queue */
   QMSG qmsg;                           /* message queue element */
   HWND hwnd, hwndClient;               /* handles for windows */

                                        /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);           /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);     /* create message queue */

                                        /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                        /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Tune", 0L, NULL, 0, &hwndClient);

                                        /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);
```

```
   WinDestroyWindow(hwnd);              /* destroy frame window */
   WinDestroyMsgQueue(hmq);             /* destroy message queue */
   WinTerminate(hab);                   /* terminate PM usage */

   return 0;
   }

CHAR stack[4096];
                                        /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   TID tid;

   switch (msg)
      {
      case WM_CREATE:
                                        /* create a new thread */
         DosCreateThread(MusicThread, &tid, stack + sizeof(stack));
         break;

      case WM_ERASEBACKGROUND:
         return TRUE;                   /* erase background */
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                         /* NULL / FALSE */
   }


VOID far MusicThread(void)
   {
   static USHORT Music[] =
                          {262,250, 262,250, 330,250, 392,250, 523,750,
                           440,1250, 440,250, 349,250, 392,250, 440,250,
                           392,1250, 262,250, 262,250, 330,250, 392,250,
                           392,750, 294,1250, 330,250, 349,250, 330,250,
                           294,250, 262,1250,
                           0,0};
   int i;

   i = 0;
   while (Music[i+1])                   /* play music */
      DosBeep(Music[i++], Music[i++]);

   }
```

```
/* ------ */
/* TUNE.H */
/* ------ */

USHORT cdecl main(void);
```

```
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);
VOID far MusicThread(void);
```

```
# --------------
# TUNE Make file
# --------------

tune.obj: tune.c tune.h
   cl -c -G2s -W3 -Zp tune.c

tune.exe: tune.obj tune.def
   link /NOD tune,,NUL,os2 slibce,tune
```

```
; --------
; TUNE.DEF
; ---------

NAME        TUNE WINDOWAPI

DESCRIPTION 'Sound a tune'

PROTMODE
STACKSIZE   4096
```

You will notice that there is an extra section in this program: the C function *MusicThread*. This function uses a simple *while* loop to play frequencies and durations from data stored in the array *Music*. To play the music we use the kernel function DosBeep.

Sound a Tone

```
USHORT DosBeep(usFrequency, usDuration)
USHORT usFrequency     Frequency of tone (Hertz)
USHORT usDuration      Duration of tone (milliseconds)

Returns: 0 if successful, error value if error (such as
         ERROR_INVALID_FREQUENCY)
```

A value of 0 for the frequency terminates the loop and ends the music.

The DosCreateThread Function

The important function in TUNE is DosCreateThread. You may have noticed that although there is a C function called *MusicThread*, there is no call

to this function. DosCreateThread executes this call, but it does more: It starts *MusicThread* running *at the same time* as the initial thread. (The initial thread is the one started when *main* is executed.)

Create a Thread

```
USHORT DosCreateThread(pfnFunction, ptidThread, pbThrdStack)
PFNTHREAD pfnFunction    Pointer to function address
PTID ptidThread          Address for thread ID
PBYTE pbThrdStack        Pointer to start of stack

Returns: 0 if successful, error value if error
```

The first argument to DosCreateThread is the address of the function that will be started as a separate thread, represented in C simply by the function name. The second argument is the address where OS/2 will return an identifier for the thread: *tid.* This identifier is not used in our example, but can be used by other functions to reference the thread.

While the C compiler takes care of providing stack space for the initial thread in an application, threads created subsequently must provide their own stack space. The third argument to DosCreateThread tells OS/2 where this stack space is. The system, as well as your application, needs to use the stack, so it should be larger than you might guess; 4K bytes is adequate in this example.

Simple Synchronization

Now that we have two threads running at the same time, we need to deal with the problem of how to synchronize their actions. In this program, synchronization between the two threads is rather elementary. It takes place at two points. The first point occurs when the initial thread starts the second thread with DosCreateThread. After this, the two threads go their separate ways. The initial thread can receive and act on messages to its heart's content. The second thread, *MusicThread,* goes merrily about the business of playing its tune.

The second point of synchronization occurs when the program terminates. If the user selects "Close" from the System menu, the application's initial thread is terminated. Terminating the initial thread automatically

terminates all the threads in the program. In this case the second thread terminates. You can verify this yourself: If the music is still playing, it stops in mid-phrase.

Doing It Wrong

You can use the TUNE example to see what happens if you write your application so that it processes a lengthy task without responding to user-related messages. To do this, change the call to DosCreateThread to a simple C language call to the *MusicThread* function. Now if you try to make menu selections or interact with the system in any other way while the music is playing, you'll find your system is locked up and unresponsive. You've changed the multithread program to a single thread program and are suffering the consequences. (You'll regain control when the music stops.)

THE SEMAPHORE SOLUTION

Our second example is similar to TUNE in that it plays a tune. However, it uses a more sophisticated method of interthread synchronization. It starts playing, not when the program is first executed, but when the user selects the "Play" option from the action bar. When the music starts, the "Play" item is disabled; when the music ends, it's enabled again.

This program demonstrates two ways that threads can communicate with each other. The first is the *semaphore,* the major OS/2 kernel mechanism used to synchronize threads. A semaphore, as its name implies, is a sort of traffic light or stop-and-go switch. In one typical situation, a thread clears a semaphore to tell a second thread to start doing something, and the second thread resets the semaphore when it's finished.

The second kind of thread communication demonstrated by this program is typical of PM, rather than the OS/2 kernel: One thread can transmit a message to the other.

Here are the listings for SYNCTUNE.C, SYNCTUNE.H, SYNCTUNE, SYNCTUNE.DEF, and SYNCTUNE.RC.

```
/* ------------------------ */
/* SYNCTUNE.C - Play a tune */
/* ------------------------ */

#define INCL_WIN
#define INCL_DOS

#include <os2.h>
```

```
#include "synctune.h"

HAB hab;                                   /* handle for anchor block */
HWND hwndClient;

USHORT cdecl main(void)
   {
   HMQ hmq;                                /* handle for message queue */
   QMSG qmsg;                              /* message queue element */
   HWND hwnd;                              /* handles for windows */

                                           /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_MENU;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);              /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);        /* create message queue */

                                           /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, CS_SIZEREDRAW, 0);

                                           /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
             szClientClass, " - Tune", 0L, NULL, ID_FRAMERC, &hwndClient);

                                           /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);                 /* destroy frame window */
   WinDestroyMsgQueue(hmq);                /* destroy message queue */
   WinTerminate(hab);                      /* terminate PM usage */

   return 0;
   }


CHAR stack[4096];
HSEM hsem;
                                           /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                  MPARAM mp2)
   {
   TID tid;
   static HWND hwndMenu;

   switch (msg)
      {
      case WM_COMMAND:
         switch (COMMANDMSG(&msg) -> cmd)
            {
            case ID_PLAY:
                                           /* disable PLAY */
               WinSendMsg(hwndMenu, MM_SETITEMATTR,
                     MPFROM2SHORT(ID_PLAY, FALSE),
                     MPFROM2SHORT(MIA_DISABLED, MIA_DISABLED));
```

```
                DosSemClear(&hsem);    /* clear semaphore */
                break;
            }
        break;

    case MTM_TUNE_END:
                                      /* enable PLAY */
        WinSendMsg(hwndMenu, MM_SETITEMATTR,
                   MPFROM2SHORT(ID_PLAY, FALSE),
                   MPFROM2SHORT(MIA_DISABLED, ~MIA_DISABLED));
        break;

    case WM_CREATE:
                                      /* get menu window handle */
        hwndMenu = WinWindowFromID(WinQueryWindow(hwnd, QW_PARENT,
                                      FALSE),FID_MENU);
        DosSemSet(&hsem);             /* set semaphore */
                                      /* create new thread */
        DosCreateThread(MusicThread, &tid, stack + sizeof(stack));
        break;

    case WM_ERASEBACKGROUND:
        return TRUE;                  /* erase background */
        break;

    default:
        return(WinDefWindowProc(hwnd, msg, mp1, mp2));
        break;
    }

   return NULL;                       /* NULL / FALSE */
   }

                                      /* music thread */
VOID far MusicThread(void)
   {
   static USHORT Music[] =
                     {262,250, 262,250, 330,250, 392,250, 523,750,
                      440,1250, 440,250, 349,250, 392,250, 440,250,
                      392,1250, 262,250, 262,250, 330,250, 392,250,
                      392,750, 294,1250, 330,250, 349,250, 330,250,
                      294,250, 262,1250,
                      0,0};
   int i;
   HAB habMT;

   habMT = WinInitialize(NULL);       /* initialize PM usage for thread */

   while (TRUE)                       /* do forever */
      {
      i = 0;
      DosSemWait(&hsem, SEM_INDEFINITE_WAIT);  /* wait for sem to clear */
      while (Music[i+1])
         DosBeep(Music[i++], Music[i++]);      /* sound music */
      DosSemSet(&hsem);                        /* set sem */
                                      /* tell client that tune ended */
      WinPostMsg(hwndClient, MTM_TUNE_END, NULL, NULL);
      }
   }
```

```
/* ---------- */
/* SYNCTUNE.H */
/* ---------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);
VOID far MusicThread(void);

#define ID_FRAMERC 1
#define ID_PLAY 100

#define MTM_TUNE_END WM_USER
```

```
# ------------------
# SYNCTUNE Make file
# ------------------

synctune.obj: synctune.c synctune.h
   cl -c -G2s -W3 -Zp synctune.c

synctune.res: synctune.rc synctune.h
   rc -r synctune.rc

synctune.exe: synctune.obj synctune.def
   link /NOD synctune,,NUL,os2 slibce,synctune
   rc synctune.res

synctune.exe: synctune.res
   rc synctune.res
```

```
; ------------
; SYNCTUNE.DEF
; ------------

NAME        SYNCTUNE WINDOWAPI

DESCRIPTION 'Sync. Tune'

PROTMODE
STACKSIZE   4096
```

```
/* ----------- */
/* SYNCTUNE.RC */
/* ----------- */

#include <os2.h>
#include "synctune.h"

MENU ID_FRAMERC
   BEGIN
      MENUITEM "~Play", ID_PLAY
   END
```

The structure of this program is basically similar to that of TUNE. The initial thread takes care of message processing, while *MusicThread* plays the music. However, the fact that the music is now started from a menu imposes the need for some form of communication between the threads. A semaphore is used by the initial thread to tell *MusicThread* when to start playing, and a message is transmitted by *MusicThread* to tell the initial thread that it is finished.

WinInitialize with Threads

As we saw in Chapter 3, any thread that intends to use PM functions (those starting with Win and Gpi) must first execute WinInitialize. We didn't do this with the child thread in TUNE, because it didn't use any PM functions. In SYNCTUNE, however, our child thread is more ambitious, so WinInitialize is the first function executed.

Semaphores

A semaphore is essentially a switch that can be set, cleared, and tested. In this program the semaphore is an external variable, *hsem*. Because it's stored in memory, it's called a *RAM semaphore* and is specified by its address. (There are also *system semaphores*, which are accessed using handles.)

In this example the semaphore is set by the initial thread during WM_CREATE processing. The function used for this is DosSemSet.

Set a Semaphore

```
USHORT DosSemSet(hsem)
HSEM hsem     Semaphore handle

Returns: 0 if successful, error value if error
```

Once the semaphore is set, the initial thread starts the child thread *MusicThread* with DosCreateThread. The child thread immediately executes another semaphore function, DosSemWait. Until the semaphore is cleared, *MusicThread* will be blocked at this API.

Wait for a Semaphore To Be Cleared

```
USHORT DosSemWait(hsem, lTimeOut)
HSEM hsem        Semaphore handle or address
LONG lTimeOut    Time out

Returns: 0 if successful, error value if error
```

The first argument to this function is the address of the semaphore. The second argument specifies the maximum time the thread will wait for the semaphore to clear. The choices are the number of milliseconds to wait, SEM_IMMEDIATE_RETURN for an immediate return, and SEM_INDEFINITE_WAIT if it should wait indefinitely. This last choice is the simplest, and we use it here. Thus *MusicThread* will be blocked until something clears the semaphore.

The semaphore is cleared as a result of the user selecting "Play" from the action bar. This causes a WM_COMMAND message with a command value of ID_PLAY to be received by the initial thread. This thread then disables the menu item and clears the semaphore with DosSemClear.

Clear a Semaphore

```
USHORT DosSemClear(hsem)
HSEM hsem Semaphore handle or address

Returns: 0 if successful, error value if error
```

As soon as the initial thread clears the semaphore, the child thread, which has been waiting on the semaphore, becomes unblocked from DosSemWait, and starts to play the tune. When it's finished, it sets the semaphore again with DosSemSet. These statements in *MusicThread* are all in an endless *while* loop, so the thread returns to the beginning of the loop, where it waits once again on the semaphore.

Notice that it is the child thread that waits on the semaphore with DosSemWait. Be careful if you turn this kind of synchronization around and have the initial, message-handling thread wait on a semaphore, because it might have to wait longer than one-tenth of a second. Only threads that don't handle messages should wait on semaphores with DosSemWait. If a message-handling thread must wait on a semaphore, it can do so with the WinMsgSemWait API, which allows the thread to receive messages while it is waiting.

You might imagine that you could let the two threads manipulate the semaphore directly, without using the Dos semaphore functions. For instance, your program could check whether the semaphore had been cleared by repeatedly executing an *if* statement. However, this violates one of the basic rules in a multitasking system: Avoid *busy-waiting*. Busy-waiting is repeatedly checking, in a loop, if an event has occurred. Such an approach wastes machine cycles that could be used by other threads and applications, and is a form of antisocial behavior in a multitasking environment. DosSemWait, on the other hand, returns control to the operating system while the application waits on the semaphore.

Interthread Messages in SYNCTUNE

When *MusicThread* has finished playing its tune, it needs to communicate this fact to the initial thread, so the initial thread can enable the "Play" menu selection. It does this—not with a semaphore—but by posting a message.

We discussed the WinPostMsg function in Chapter 5. Here we use it to post an MTM_TUNE_END message (defined in SYNCTUNE.H) to the client window procedure in the initial thread. When the initial thread receives this message, it enables the "Play" menu item. Now the user can play the tune again by selecting the menu item.

If you needed to transfer additional information from one thread to another, you could use the *mp1* and *mp2* message parameters in WinPostMsg; we don't need them in this example, so they're set to NULL.

Notice that the music thread in SYNCTUNE doesn't create a message queue or a message loop. It doesn't need these features, because it doesn't receive any messages, although it posts a message to the client window in the other thread.

THE OBJECT WINDOW SOLUTION

Let's look at another way to solve the problem of carrying out a lengthy task and responding to messages at the same time. Our example program, OBJTUNE, uses two threads as the previous examples did, but here the second thread creates a new kind of window called an *object window*.

Object Windows

As discussed in Chapter 4, all windows in PM are really *objects*. Objects are programming entities that carry out specific tasks and use messages to communicate with other objects. However, the windows that we have

examined so far in PM are not only objects in this sense. They also have a visual aspect: They manifest themselves on the screen. A scroll bar window creates a scroll bar that can be seen, for example. Even a client window is visual, in the sense that it's a blank area on which the client window procedure can display whatever it wants.

An object window, on the other hand, is a window with no visual aspect. Its purpose is not to display anything, but to carry out other kinds of programming tasks. Because object windows are non-visual, they don't need to process user-related messages like WM_CHAR or WM_MOUSEMOVE, or display-oriented messages like WM_SIZE and WM_PAINT. This leaves them free to carry out any task assigned to them by the programmer. The programmer defines the messages going to an object window, and the messages that it sends. An object window is very much an *object* in the sense that the word is used in object-oriented programming.

Object-Oriented Programming

As we've seen, programming the PM user interface is based on an object-oriented model. Visual windows like scroll bars, menus, and the client window are objects and communicate by passing messages back and forth. You can use this object-oriented model for the user interface but write the rest of the program using the traditional procedural model. However, there is a more interesting and consistent approach to PM programming. Object windows allow the programmer to use an object-oriented approach not just for the user interface, but for other aspects of the program as well.

This example uses an object window and two threads to solve the problem of responding to messages in a timely fashion. However, this does not imply that object windows are useful only in this context. They can be used to implement any programming task. An entire application can be written in object-oriented form, implementing programmer-defined objects as object windows. For instance, one object window could read and write disk files, another could sort records from the files, and a third could generate reports from these records. Our OBJTUNE program provides a small example of how this approach is used.

The OBJTUNE Example

The example program plays the same song as do TUNE and SYNCTUNE, and adds two other musical selections as well.

Here are the listings for OBJTUNE.C, OBJTUNE.H, OBJTUNE, OBJTUNE.DEF, and OBJTUNE.RC:

```
/* ---------------------- */
/* OBJTUNE.C  Play a tune */
/* ---------------------- */

#define INCL_WIN
#define INCL_DOS

#include <os2.h>

#include "objtune.h"

HWND static hwndClient;

USHORT cdecl main(void)
   {
   HAB hab;                              /* handle for anchor block */
   HMQ hmq;                              /* handle for message queue */
   QMSG qmsg;                            /* message queue element */
   HWND hwnd;                            /* handles for windows */

                                         /* create flags */
   ULONG flCreateFlags = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                         FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST |
                         FCF_MENU;

   static CHAR szClientClass[] = "Client Window";

   hab = WinInitialize(NULL);            /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);      /* create message queue */

                                         /* register client window class */
   WinRegisterClass(hab, szClientClass, ClientWinProc, 0L, 0);

                                         /* create standard window */
   hwnd = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flCreateFlags,
            szClientClass, " - Music Object", 0L, NULL, ID_FRAMERC,
            &hwndClient);

                                         /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwnd);               /* destroy frame window */
   WinDestroyMsgQueue(hmq);              /* destroy message queue */
   WinTerminate(hab);                    /* terminate PM usage */

   return 0;
   }

static USHORT fPlay;

                                         /* client window procedure */
MRESULT EXPENTRY ClientWinProc(HWND hwnd, USHORT msg, MPARAM mp1,
                     MPARAM mp2)
   {
```

```
static TID tid;
static HWND hwndMenu;
static CHAR stack[4096];
static HWND hwndObject = NULL;

switch (msg)
   {
   case WM_COMMAND:
      switch (COMMANDMSG(&msg) -> cmd)
         {
         case ID_SONG1:
         case ID_SONG2:
         case ID_SONG3:
                                    /* disable PLAY */
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
                  MPFROM2SHORT(ID_PLAY_SUBMENU, FALSE),
                  MPFROM2SHORT(MIA_DISABLED, MIA_DISABLED));
                                    /* enable STOP */
            WinSendMsg(hwndMenu, MM_SETITEMATTR,
                  MPFROM2SHORT(ID_STOP, FALSE),
                  MPFROM2SHORT(MIA_DISABLED, ~MIA_DISABLED));
                                    /* tell object to play song */
            WinPostMsg(hwndObject, MTM_PLAY,
                  MPFROMSHORT(SHORT1FROMMP(mp1) - ID_SONG1),
                  NULL);
                  break;

         case ID_STOP:
            fPlay = FALSE;          /* stop playing music */
            break;
         }
      break;

   case MTM_TUNE_END:
                                    /* disable STOP */
      WinSendMsg(hwndMenu, MM_SETITEMATTR,
                  MPFROM2SHORT(ID_STOP, FALSE),
                  MPFROM2SHORT(MIA_DISABLED, MIA_DISABLED));
                                    /* enable PLAY */
      WinSendMsg(hwndMenu, MM_SETITEMATTR,
                 MPFROM2SHORT(ID_PLAY_SUBMENU, FALSE),
                 MPFROM2SHORT(MIA_DISABLED, ~MIA_DISABLED));
      break;

   case MTM_READY:
      hwndObject = HWNDFROMMP(mp1);    /* get object handle */
                                       /* enable PLAY */
      WinSendMsg(hwndMenu, MM_SETITEMATTR,
                 MPFROM2SHORT(ID_PLAY_SUBMENU, FALSE),
                 MPFROM2SHORT(MIA_DISABLED, ~MIA_DISABLED));
      break;

   case WM_CREATE:
                                    /* get menu window handle */
      hwndMenu = WinWindowFromID(WinQueryWindow(hwnd, QW_PARENT,
                                    FALSE),FID_MENU);
                                    /* create music thread */
      DosCreateThread(MusicThread, &tid, stack + sizeof(stack));
      break;

   case WM_ERASEBACKGROUND:
```

```
         return TRUE;                  /* erase background */
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                        /* NULL / FALSE */
   }

                                       /* music thread */
VOID far MusicThread(void)
   {
   HAB hab;                            /* handle for anchor block */
   HMQ hmq;                            /* handle for message queue */
   QMSG qmsg;                          /* message queue element */
   HWND hwndObject;

   static CHAR szClientClass[] = "Music Object";

   hab = WinInitialize(NULL);          /* initialize PM usage */
   hmq = WinCreateMsgQueue(hab, 0);    /* create message queue */

                                       /* register client window class */
   WinRegisterClass(hab, szClientClass, MusicObject, 0L, 0);

                                       /* create object window */
   hwndObject = WinCreateWindow(HWND_OBJECT, szClientClass, NULL, 0L,
                   0, 0, 0, 0, NULL, HWND_TOP, 0, NULL, NULL);

                                       /* messages dispatch loop */
   while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
      WinDispatchMsg(hab, &qmsg);

   WinDestroyWindow(hwndObject);       /* destroy window */
   WinDestroyMsgQueue(hmq);            /* destroy message queue */
   WinTerminate(hab);                  /* terminate PM usage */
   }

                                       /* object window procedure */
MRESULT EXPENTRY MusicObject(HWND hwnd, USHORT msg, MPARAM mp1,
                    MPARAM mp2)
   {
   static USHORT aMusic[][CMAXSONGLEN] =
                  {{262,250, 262,250, 330,250, 392,250, 523,750,
                    440,1250, 440,250, 349,250, 392,250, 440,250,
                    392,1250, 262,250, 262,250, 330,250, 392,250,
                    392,750, 294,1250, 330,250, 349,250, 330,250,
                    294,250, 262,1250,
                    0,0},
                   {292,500, 440,500, 349,750, 330,250, 294,250,
                    349,250, 330,250, 294,250, 278,250, 330,250,
                    220,1000, 294,250, 220,250, 330,250, 220,250,
                    349,250, 330,125, 294,125, 330,250, 220,250,
                    294,250, 220,250, 330,250, 220,250, 349,250,
                    330,125, 294,125, 330,250, 220,250, 294,1000,
                    0,0},
                   {440,250, 392,250, 349,250, 349,500, 349,250,
                    349,250, 349,250, 262,250, 220,250, 234,250,
                    262,250, 262,500, 262,250, 262,500, 349,250,
```

```
                        392,250, 440,500, 440,250, 440,250, 440,250,
                        440,250, 392,250, 349,250, 392,250, 440,250,
                        392,500, 392,250, 392,1000,
                        0,0}};
   int i = 0;
   int iSong;

   switch (msg)
      {
      case MTM_PLAY:
         fPlay = TRUE;                 /* flag true */
         iSong = SHORT1FROMMP(mp1);    /* get song no. */
         while (aMusic[iSong][i+1] && fPlay)    /* while flag tune */
                                                /* and song not over */
            DosBeep(aMusic[iSong][i++], aMusic[iSong][i++]);    /* play */
                                       /* tell client: tune ended */
         WinPostMsg(hwndClient, MTM_TUNE_END, NULL, NULL);
         break;

      case WM_CREATE:
                                       /* tell client: object is ready */
         WinSendMsg(hwndClient, MTM_READY, MPFROMHWND(hwnd), NULL);
         break;

      default:
         return(WinDefWindowProc(hwnd, msg, mp1, mp2));
         break;
      }

   return NULL;                        /* NULL / FALSE */
   }
```

```
/* --------- */
/* OBJTUNE.H */
/* --------- */

USHORT cdecl main(void);
MRESULT EXPENTRY ClientWinProc(HWND, USHORT, MPARAM, MPARAM);
VOID far MusicThread(void);
MRESULT EXPENTRY MusicObject(HWND, USHORT, MPARAM, MPARAM);

#define ID_FRAMERC 1
#define ID_PLAY_SUBMENU 10
#define ID_SONG1 101
#define ID_SONG2 102
#define ID_SONG3 103
#define ID_STOP 200

#define MTM_READY WM_USER
#define MTM_PLAY WM_USER + 1
#define MTM_TUNE_END WM_USER + 2

#define CMAXSONGLEN 100
```

```
# -----------------
# OBJTUNE Make file
# -----------------

objtune.obj: objtune.c objtune.h
```

```
21   cl -c -G2s -W3 -Zp objtune.c

objtune.res: objtune.rc objtune.h
   rc -r objtune.rc

objtune.exe: objtune.obj objtune.def
   link /NOD objtune,,NUL,os2 slibce,objtune
   rc objtune.res

objtune.exe: objtune.res
   rc objtune.res
```

```
; -----------
; OBJTUNE.DEF
; -----------

NAME        OBJTUNE WINDOWAPI

DESCRIPTION 'Sound a tune'

PROTMODE
STACKSIZE   4096
```

```
/* ---------- */
/* OBJTUNE.RC */
/* ---------- */

#include <os2.h>
#include "objtune.h"

MENU ID_FRAMERC
   BEGIN
      SUBMENU "~Play", ID_PLAY_SUBMENU, ,MIA_DISABLED
         BEGIN
            MENUITEM "~Song1", ID_SONG1
            MENUITEM "~Song2", ID_SONG2
            MENUITEM "~Song3", ID_SONG3
         END
      MENUITEM "~Stop", ID_STOP, , MIA_DISABLED
   END
```

There are two top-level menu items in OBJTUNE: "Play" and "Stop". When "Play" is selected, a submenu appears with three items: "Song1", "Song2", and "Song3", as shown in Figure 16-1. Selecting one of these items causes one of three tunes to be played. While a tune is playing, the "Play" item is disabled and is displayed in gray. If, while a tune is being played, the user selects "Stop", the tune will stop.

Overall Structure of OBJTUNE

The initial thread in OBJTUNE consists of the *main* procedure and a client window procedure. The *main* procedure creates a standard window with a

client window and sets up a message queue and message loop in the usual way. The client window procedure, during WM_CREATE processing, creates a second thread, called *MusicThread,* using DosCreateThread.

An object window must be prepared to receive messages. The second thread must therefore create a message queue and a message loop. It must also create the object window. *MusicThread* creates this window using WinCreateWindow with its *hwndParent* argument set to HWND_OBJECT. The *pszClass* and *pszTitle* arguments are filled in, as is *hwndInsertBehind.* The *id* argument can be set to a window ID number. All other arguments to WinCreateWindow are set to 0 or NULL.

As in TUNE and SYNCTUNE, the initial thread in OBJTUNE handles messages, and the second thread plays the music. However, in OBJTUNE the presence of the object window allows the two threads to communicate using messages in a more sophisticated way than in the previous examples. Instead of the messages all going one way, the threads can send messages back and forth.

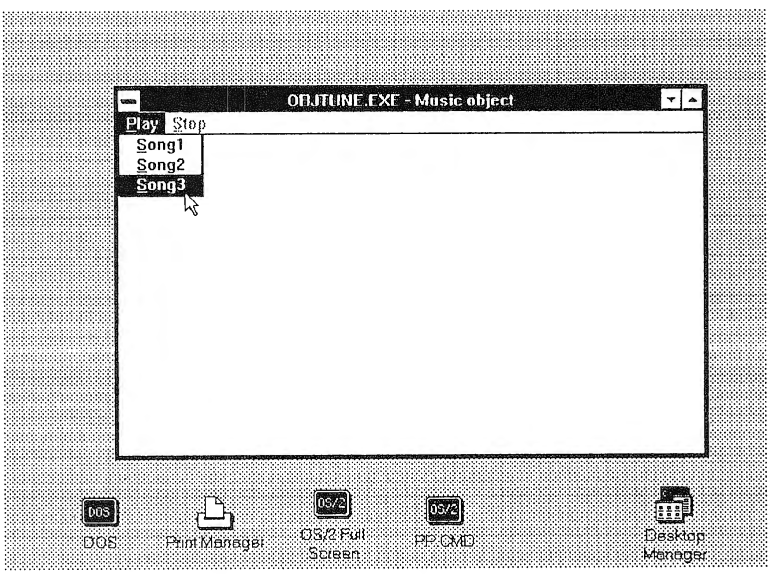


Figure 16-1. Output of the OBJTUNE program

Message Flow in OBJTUNE

We'll look at the message flow in OBJTUNE from an object-oriented perspective. The first object is the client window in the first thread. This object is mostly involved with the user interface—in this case, dealing with menu selections—so we'll call it the *interface object.* The second object is the object window in the second thread. It plays the tunes, so we'll call it the *music object.* We'll focus on each of these objects in turn.

The Interface Object

The interface object acts on four messages: WM_CREATE, MTM_READY, WM_COMMAND, and MTM_TUNE_END.

When the interface object receives a WM_CREATE message, it executes DosCreateThread to start the second thread, which will create the music object.

When it receives an MTM_READY message (*#define*d in OBJTUNE.H), the interface object knows that the music object is ready, so it enables the "Play" menu. The "Play" menu is disabled (gray) when the program first starts because the music object is not yet created, and it can't play anything until it exists.

When the user selects "Song1", "Song2", or "Song3" from the "Play" menu, the interface object receives a WM_COMMAND message. Upon receipt of this message it posts an MTM_PLAY message to the music object to tell it to start the music. It also disables the "Play" menu and enables the "Stop" menu.

How does the interface object tell the music object which of the three tunes to play? This information is transferred in the *mp1* parameter of the MTM_PLAY message. This parameter is given the value

```
MPFROMSHORT(SHORT1FROMMP(mp1) - ID_SONG1)
```

SHORT1FROMMP(mp1) is the command value sent with WM_COMMAND; it's ID_SONG1, ID_SONG2, or ID_SONG3. By subtracting ID_SONG1, the song number is obtained.

Upon receiving the MTM_TUNE_END message, the interface object knows that the tune is finished, so it disables the "Stop" menu and enables the "Play" menu so the user can select another tune.

The Music Object

The music object acts on only two messages: WM_CREATE and MTM_PLAY.

When the music object is created, it receives a WM_CREATE message, and on receipt of this message it sends an MTM_READY message to the interface object to signify that it's ready to play.

When it receives the MTM_PLAY message, the music object sets *iSong*, the song number, to the value in *mp1*, which determines which song to play. It then plays the tune. When it finishes, it posts an MTM_TUNE_END message to the interface object.

The interchange of messages between the two objects is shown in Figure 16-2.

The inter-object communication concerning the "Stop" menu item is not implemented using messages, but with a simple external variable. This variable is called *fPlay*, and its role is to stop the song in midstream if the user selects "Stop".

Before starting the song, the music object sets *fPlay* to TRUE. In the *while* loop, while it plays the music, it checks the status of *fPlay*. If the user selects "Stop", the interface object sets *fPlay* to FALSE. The music object will learn this on completion of the current musical note, the *while* loop will terminate, and the music object will post the MTM_END_TUNE message to the interface object, signaling that it's done.

Serially Reusable Resources

Some system resources—such as the printer—can be used by only one thread at a time. Such resources are called serially reusable resources (SRRs). When a thread uses an SRR, and a second thread (from the same or a different program) attempts to use the same SRR, the system will make the second thread wait until the first one is finished.

Selecting a disabled menu item ordinarily sounds a beep. However, since the system speaker is an SRR, if you select a disabled menu item while a song is playing, you will only hear the warning beep after the song is finished.

The issues of task synchronization and SRR management require special consideration when programming in a multitasking environment. These issues are related more to the OS/2 kernel than to PM, and so will not be discussed further here.

THE FUTURE

This chapter has brought together some of PM's most powerful features. The examples demonstrate not only the user interface and multitasking, but the object-oriented architecture underlying PM. In the next few years, applications and programming languages will make increasing use of this object-oriented architecture. Object-oriented programming offers many advantages in the conceptualization, development, and maintenance of software applications. Also, multiprocessing hardware will allow separate

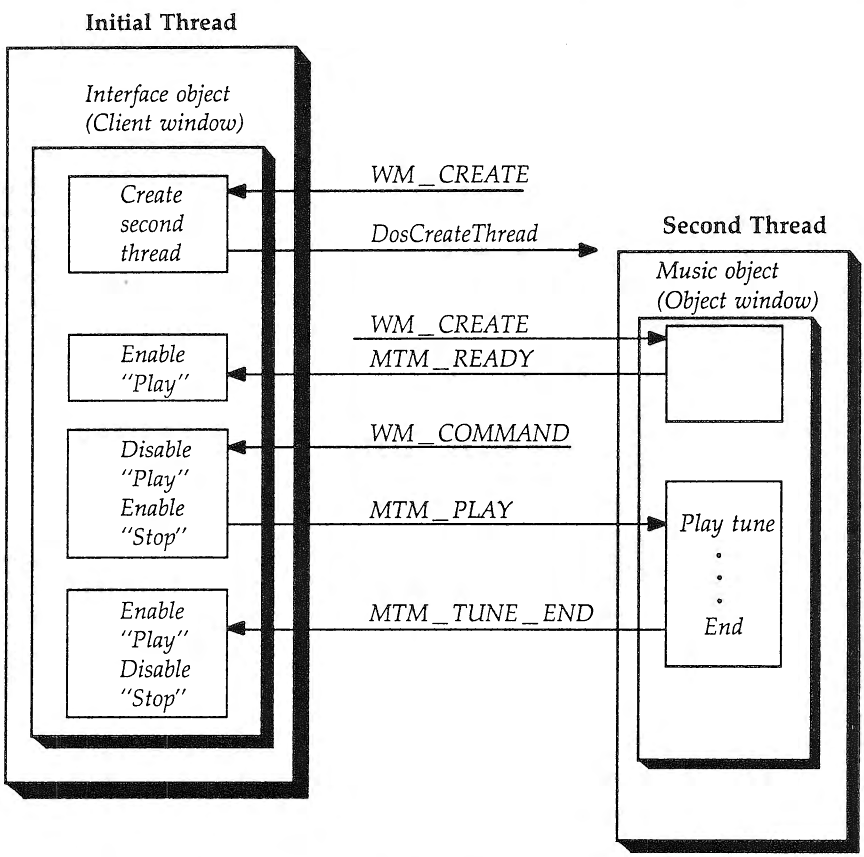


Figure 16-2. Message passing in OBJTUNE

threads and processes actually to run simultaneously, rather than alternating machine cycles. This will provide major performance increases to PM applications.

With your knowledge of PM programming you are in a position to take advantage of these developments, and to create applications that are more powerful and easier to use than anything dreamt of just a few years ago.

In the next (and final) chapter we'll discuss some points of interest to programmers who plan to adapt their applications to an international market.

17

FOREIGN LANGUAGE SUPPORT

PM contains several features that make it easier to create versions of an application for different national languages. That is, if you've written an application for the United States market, you can adapt it to the French, German, or even Russian or Japanese markets by making a few well-defined changes. We touched on some of these changes earlier; in this chapter we'll review our earlier comments and briefly mention some additional topics involved in foreign language support.

RESOURCES AND DYNAMIC LINKING

All text—such as instructions to the user and the names of controls and menu items—should be placed in resources. If this is done, then changing the text to another language requires only that the resource files be recompiled. The source code need not be changed or recompiled.

Some icons may also be country-dependent and can profit in the same way from being placed in resource files. For example, an icon representing a mailbox might need to be changed to reflect the appearance of typical mailboxes in the relevant country. Icons that include text should also be modified. Resources are discussed in Chapter 7.

Placing all language-dependent data in dynamic link libraries (DLLs) is helpful in creating foreign language versions of programs. A different version of the application may require new DLLs, but the executable files for the program need not be altered. This can simplify manufacturing, distribution, and maintenance of the program, since there is no necessity for multiple versions of the source code or executables.

CODE PAGES

What happens if a foreign language uses characters that aren't available in the language you used originally? You might have a version of your application running in English, and want to modify it for Greek, Hebrew, or Russian, for example. The answer to this is to use a different *code page.* A code page is used to translate between the computer's binary representation of a character and the character that will be displayed on the device. For the keyboard, it specifies what character code the application will receive for each key pressed by the user.

Each device has its own code page. On some devices, such as printers that do not support downloadable fonts, the code page is part of the device's hardware (ROM) and can't be changed. Smarter devices allow the application to change their code page.

In the code page the visual representation of the character is called a *glyph,* and its numerical value is called a *code point* or *character value.* The code page might specify that the glyph 'A', for example, has the character value 65 (decimal). A code page typically contains 256 glyphs, although it may contain more.

Each code page is given a number called a *code page identifier.* This is used when the code page is identified in API calls. For example, as we saw in Chapter 11, the code page must be specified when a logical font is created with GpiCreateLogFont.

Currently the following code pages are available for PM:

Number	Code Page
437	United States (Original PC ASCII)
500	EBCDIC International version
850	Multilingual
863	French-Canadian
865	Nordic

Code page 437 corresponds to the old PC extended character set used in MS-DOS. All system text in the current version of OS/2 is written to be used with code page 850, which contains all the characters necessary for English and the major European languages, including French, Italian, German, Dutch, Swedish, and so on. Code page 850 is similar to 437, but substitutes foreign language characters (usually accented characters) for many of the graphics symbols.

You can set a new code page with GpiSetCp and find out the current code page with GpiQueryCp. The function WinQueryCpList retrieves a list of all the code pages available in the system. Kernel functions, like DosSetCp, are also available for these tasks.

Formatting Information

OS/2 maintains *formatting information* for each country. This information specifies the formats used to write dates and times, monetary values, and numbers. For example, in France the month is written second in expressions like 20/12/92, and the decimal point in numbers is often represented by a comma, while the thousands separator is a period.

The formatting information is stored in the COUNTRY.SYS file, and can be retrieved with the DosGetCtryInfo function. It is stored in a structure of type COUNTRYINFO, which looks like this:

```
struct _COUNTRYINFO  {
   USHORT country;                  /* country code (France=033) */
   USHORT codepage;                 /* reserved; must be 0 */
   USHORT fsDateFmt;                /* DATEFMT_MM_DD_YY, etc. */
   CHAR szCurrency[5];              /* ASCIIZ string: "$", etc. */
   CHAR szThousandSeparator[2];     /* ASCIIZ string: ",", etc. */
   CHAR szDecimal[2];               /* ASCIIZ string: ".", etc. */
   CHAR szDateSeparator;            /* ASCIIZ string: "/", etc. */
   CHAR szTimeSeparator;            /* ASCIIZ string: ":", etc. */
   UCHAR fsCurrencyFmt;             /* CURRENCY_FOLLOW, etc. */
   UCHAR cDecimalPlace;             /* decimal places in currency value */
   UCHAR fsTimeFmt;                 /* 0x0001=24 hours, else 12 hours */
   USHORT abReserved1[2];           /* reserved; must be 0 */
   CHAR szDataSeparator[2];         /* ASCIIZ string */
   USHORT abReserved2[5];           /* reserved; must be 0 */
} COUNTRYINFO;
```

The Country Code

The first member of the COUNTRYINFO structure is the country code. It can be one of the following:

Country Code	Country
001	United States
002	Canada (French)
003	Latin America
031	Netherlands
032	Belgium
033	France
034	Spain
039	Italy
041	Switzerland
044	United Kingdom
045	Denmark
046	Sweden
047	Norway
049	Germany
061	Australia
351	Portugal
358	Finland
972	Israel

(These codes correspond to the country code used in international telephone calls. You dial 33 to get France, for example.)

Code Page Translation

It may happen that you're working with text based on one code page, but need to print or display the text on a device that uses another code page. For instance, your printer might use a different code page from the rest of your system. Several PM functions help solve this problem. WinCpTranslate-Char translates a single character from one code page to another, while WinCpTranslateString does the same for an entire string.

The Collating Sequence

Another aspect of different character sets is the *sorting weight.* This is a number applied to each character to determine where the character falls in an alphabetized list. The list of sorting weights is called the *collating sequence table.* This table is specific to a particular code page and country code. When text is sorted, the sorting weights in the collating sequence table are com-

monly used. The OS/2 SORT function sorts text this way, for example. The collating sequence does not necessarily correspond to the character values of the characters. The letter 'é' ('e' with an accent) might follow an unaccented 'e' in the collating sequence, but have a completely different character value.

The collating sequence table is an array with 256 elements, each of which contains the sorting weight of the corresponding character. The DosGetCollate function retrieves the collating table for a given country code and code page identifier.

Code-Page-Sensitive Functions

Some of the text manipulation functions of current C compilers are based on the old code page 437, and may not work correctly with the new code pages. These functions involve case conversions and sorting. PM provides its own functions to avoid these problems. WinUpperChar changes the case of single characters, WinUpper changes the case of text strings, and WinCompareStrings compares two strings alphabetically based on the collating sequence.

ACCENTED CHARACTERS

English doesn't use accented characters, but many other languages—such as French, German, and Spanish—do. Examples are ü, é, â, à, and ç. Accented characters present special problems for input.

Usually the keyboard is specific to the language used in a particular system. Typing accented characters depends on the keyboard. On some systems there is a keyboard key called a *dead key.* To type a character with an accent mark, the user first types the dead key, then the accent mark, and finally the character itself. On other systems a single key may generate the accent mark, and a second key generates the character.

A PM application processes accented characters, like other keyboard characters, upon receipt of the WM_CHAR message. If the character is an accent, the KC_DEADKEY flag will be set. The application should display the accent character, but not advance the cursor. The next WM_CHAR message has the KC_COMPOSITE bit set and contains the character itself. It is displayed and the cursor is advanced normally.

The display of accented characters is not usually a problem, since the character including the accent is represented by a glyph in the code page, just as normal characters are.

TRADEMARKS

AT®	International Business Machines Corporation
CodeView™	Microsoft Corporation
dBASE IV®	Ashton-Tate
Helvetica®	Linotype Corporation
Hercules®	Hercules Computer Technology
IBM®	International Business Machines Corporation
Intel®	Intel Corporation
Lotus® 1-2-3®	Lotus Development Corporation
Macintosh®	Apple Computing, Inc.
Microsoft®	Microsoft Corporation
MS-DOS®	Microsoft Corporation
OS/2®	International Business Machines Corporation
Presentation Manager™	International Business Machines Corporation
QuickDraw™	Apple Computing, Inc.
Times®	Linotype Corporation
UNIX®	The Xerox Corporation
WordPerfect®	WordPerfect Corporation

INDEX

G

H

I

J

N

O

P

Q

R

T

U

V

W

X

Y

The manuscript for this book was prepared and submitted to Osborne/McGraw-Hill in electronic form. The acquisitions editor for this project was Jeffrey Pepper, the technical reviewer was Scott Ludwig, and the project editor was Dusty Bernard.

Text design by Marcela Hancik, using Palatino for text body and for display.

Cover art by Graphic Eye, Inc. Color separation and cover supplier, Phoenix Color Corporation. Screens produced with InSet from Inset Systems, Inc. Book printed and bound by R.R. Donnelley & Sons Company, Crawfordsville, Indiana.

OS/2®
Presentation Manager
PROGRAMMING PRIMER

Acclaim for other books by Robert Lafore:

"An excellent introduction…"
JOHN DVORAK
San Francisco Chronicle

"…A great book for programmers…"
BYTE

OS/2® Presentation Manager Programming Primer is a step-by-step guide to Presentation Manager programming. With this primer, you'll soon be writing programs that include menus, dialog boxes, and other features of the PM graphical user interface. You'll also learn about the powerful graphics functions built into PM.

Experts Asael Dror and Robert Lafore use short, clear program examples to demonstrate PM's features. No previous OS/2® programming experience is required—only a knowledge of the C language. Early chapters cover the basics, including program development, windows, and messages. You'll learn to create:

- Menus
- Buttons
- Icons
- Resources

The primer then explores PM graphics. You'll discover how to:

- Draw pictures
- Animate graphics objects
- Display text in different fonts
- Store pictures to disk
- Move and zoom pictures

Later chapters cover multitasking, the clipboard, and foreign language support.

With the skills you'll learn from Dror and Lafore, the powerful user interface and graphics features of the Presentation Manager will be yours to command.

Why This Book Is For You—See Page 1

Asael Dror is the Accelerators Software Manager at Chips and Technologies, Inc., leading the development of OS/2 Presentation Manager system software. He has been programming since 1976, is the author of an OS/2 kernel book, and the developer of an OS/2 Charter Application.

Robert Lafore has been a programmer since 1965 and has been writing programming books for ten years. His many titles, including C and assembler primers, and an OS/2 book co-authored with Peter Norton, have sold hundreds of thousands of copies.

ISBN 0-07-881467-7
52895
9 780078 814679

$28.95